

Санкт-Петербургский государственный университет информационных
технологий, механики и оптики

Кафедра “Компьютерные технологии”

П. И. Фельдман

**Разработка средств для отладки автоматных
программ, построенных на основе предложенной
библиотеки**

Пояснительная записка к бакалаврской работе

Руководитель – докт. техн. наук, профессор Шалыто А.А.

Санкт-Петербург

2005

Аннотация.....	6
ВВЕДЕНИЕ	7
1. Анализ существующих средств.....	9
1.1. Средства отладки.....	9
1.2. Автоматные библиотеки	10
1.3. Языковые средства	10
2. Библиотека <i>AFC</i>	12
2.1. Назначение.....	12
2.2. Внутреннее устройство	13
2.2.1. Диспетчер автоматов.....	13
2.2.2. Автомат	13
2.2.3. <i>ChangeEventRaiser</i>	14
2.2.4. Схема автомата	15
2.2.5. <i>DescriptedAutomatAttribute</i>	15
2.2.6. Массив чисел.....	15
2.2.7. Схема состояния	16
2.2.8. Схема перехода.....	16
2.2.10. Схемы остальных компонент автомата	16
2.2.11. Вычислитель входной переменной.....	17
2.2.12. Исполнитель выходного воздействия.....	17
2.2.13. Составитель протокола.....	17
2.2.14. Простой автомат	18
2.3. Принципы работы.....	18
2.3.1. Схема	18
2.3.2. Взаимодействие	20
2.3.3. Работа автомата	21
2.3.4. Протоколирование	22
2.3.5. Условия переходов	22
2.3.6. Исключительные ситуации	23
2.4. Использование библиотеки.....	24
Отличия библиотеки от предыдущей версии	27
3. Демонстрационное приложение	28

3.1. Постановка задачи.....	28
3.2. Словесное описание предложенного алгоритма.....	29
3.3. Автомат A1 («Путешественник»).....	30
3.3.1. Словесное описание.....	30
3.3.2. Схема связей.....	31
3.3.3. Граф переходов.....	32
3.4. Автомат A2 («Компас»).....	33
3.4.1. Словесное описание.....	33
3.4.2. Схема связей.....	33
3.4.3. Граф переходов.....	34
3.5. Работа приложения.....	34
3.6. Внешний вид приложения.....	36
4. Автоматная система отладки.....	38
4.1. Принципы работы.....	38
4.2. Структура системы отладки.....	39
4.2.1. Основной класс системы отладки.....	39
4.2.2. Классы, реализующие интерфейс системы отладки.....	41
4.3. Работа системы отладки.....	42
4.3.1. Открытие приложения, ввод параметров, запуск, остановка, прерывание.....	42
4.3.2. Наблюдение за автоматами.....	43
4.3.3. Точки остановки.....	43
4.3.4. Работа с автоматами по шагам.....	44
4.4. Внешний вид системы отладки.....	45
4.5. Пример отладки.....	50
ЗАКЛЮЧЕНИЕ.....	53
Список литературы.....	55
Приложение 1. Классы схемы автомата и схем компонентов автомата.....	56
Класс Automatscheme.....	56
Класс ActionScheme.....	64
Класс EventScheme.....	65
Класс StateScheme.....	65
Класс TransitionScheme.....	67
Класс VariableScheme.....	70
Приложение 2. Метод, реализующий переход автомата.....	70

Приложение 3. Интерфейсы «вычислителя входных переменных» и «выполнителя выходных воздействий».....	71
Интерфейс InputVariableCalculator.....	71
Интерфейс IOutputActionPerformer	72
Приложение 4. Методы класса <i>Automat</i> для работы с «вычислителями» и «выполнителями».....	73

Аннотация

В настоящей работе представлено обоснование целесообразности создания автоматной системы отладки, позволяющей программисту на время поиска ошибок и проверки работоспособности приложения, написанного с использованием автоматного подхода, абстрагироваться от средств языка, благодаря которым в программе были реализованы автоматы, и сосредоточиться на их работе.

Предлагаемая система позволяет в полной мере наблюдать за работой автоматов, вести структурное протоколирование и расставлять "точки останова" (*breakpoints*), позволяющие приостановить работу приложения в интересующем разработчика месте. Также создано приложение, на примере которого продемонстрирована функциональность предложенной системы отладки. Таким образом, настоящая работа позволяет применять автоматный подход не только при программировании, протоколировании и документировании, но и при отладке.

ВВЕДЕНИЕ

При создании и отладке приложений, используя процедурное программирование, разработчик избавлен от необходимости знать, каким образом происходит низкоуровневое взаимодействие частей программы, например вызов функции и передача параметров. Объектно-ориентированное программирование позволяет абстрагироваться от таких понятий, как функция и переменная, и мыслить в терминах классов, их методов и членов. Автоматное программирование [1], ввиду отсутствия соответствующих инструментов современных языков, вынуждает разработчика создавать автоматы на основе других средств языка. Однако отладка разрабатываемого приложения стандартными средствами, зачастую встроенными в среду разработки и которые обычно нерасширяемы, заставляет разработчика рассматривать реализацию автомата не только с позиции автоматного программирования, но и в терминах тех средств, на основе которых был создан автомат.

Цель настоящей работы исследовать современные средства отладки приложений и создать систему отладки автоматных программ, которая являлась бы аналогичной существующим системам и позволяла бы на время отладки абстрагироваться от средств языка, благодаря которым были реализованы автоматы.

Ясно, что невозможно построить систему отладки, взаимодействующую с автоматами, реализованными произвольно. Было принято решение предложить библиотеку классов, позволяющую создавать на ее основе автоматы, и разработать средство для отладки автоматов, построенных с помощью указанной библиотеки. Для определения достаточного набора инструментов, используемых для построения автоматов, было проведено исследование существующих способов создания автоматов.

Анализ существующих средств отладки, методов реализации автоматов и языков программирования для написания отладочной системы представлен в разд. 1. Описание библиотеки и предоставляемых ею возможностей приведено в

разд. 2. Для примера работы системы отладки создано демонстрационное приложение на основе упомянутой библиотеки, описание которого находится в разд. 3. В разд. 4 представлено описание системы отладки, удовлетворяющей перечисленным требованиям.

Разрабатываемая система позволяет в полной мере наблюдать за работой автоматов, вести структурное протоколирование и расставлять "точки останова" (*breakpoints*), позволяющие приостановить работу приложения в интересующем разработчика месте. Также создано приложение, на примере которого продемонстрирована функциональность предложенной системы отладки. Таким образом, настоящая работа позволяет применять автоматный подход не только при программировании, протоколировании документировании, но и при отладке.

Библиотека, демонстрационное приложение и система отладки написаны на языке *C#* под платформу *.NET Framework* и для корректной работы требуют наличие операционной системы *Windows 9x/ME/2000/XP*.

1. Анализ существующих средств

1.1. Средства отладки

Среди распространенных средств разработки приложений наибольшей популярностью пользуются *Visual Studio* и *Delphi*. Эти (а также и другие) среды предлагают исчерпывающий набор инструментов отладки. Чаще всего используются такие инструменты, как точки останова — *breakpoints*, просматриваемые выражения — *watch expressions* и стек вызовов — *call stack*. Также при отладке применяется выполнение программы по шагам. Подробные пояснения относительно упомянутых средств можно найти в работе [2].

Применительно к автоматному программированию точки останова имеют то же значение, что и в общем случае, однако целесообразна их установка на компоненты автомата — состояния, события, входные переменные, выходные воздействия и переходы. Установка точки останова на соответствующий элемент означает остановку выполнения приложения в тот момент, когда какое-либо действие происходит с выбранным компонентом.

Просматриваемые выражения могут быть представлены в виде отображения текущего состояния автоматов и их частей, а также отображением вычисленных значений входных переменных.

Стек вызовов аналогичен протоколу. Выполнение программы по шагам аналогично остановке приложения при каждом действии, произведенном с выбранным либо произвольным автоматом.

На сегодняшний день наблюдается крайний дефицит средств отладки автоматных приложений, позволяющих абстрагироваться от реализации автомата. В работе [3] предлагается использовать для отладки протокол работы автомата, выполненный в автоматных терминах. Однако в случае большого протокола становится затруднительно локализовать ошибку.

Протоколирование всех элементарных действий, произведенных с автоматом (начало и конец вычисления значения входной переменной, начало и

конец выполнения выходного воздействия, начало и конец проверки условия перехода и т.д.) неоправданно увеличивает размер протокола. Исключение такой информации из протокола может привести к усложнению поиска ошибки и даже невозможности ее локализации.

В данной работе разработаны и предложены средства отладки автоматных приложений, аналогичные вышеперечисленным инструментам популярных сред разработки.

1.2. Автоматные библиотеки

За время существования автоматного подхода был предложен ряд методов реализации автоматов (функции, используемые в структурном программировании [3], методы классов [4], наследование от класса, реализующего базовую функциональность автомата [5] и т.д.). Подробное перечисление методов создания автоматов можно найти в работе [6].

Отметим, что при этом были разработаны объектно-ориентированные библиотеки. Создание автоматов на основе объектной библиотеки не только позволяет свести к минимуму издержки, связанные с отсутствием автоматных средств в языках программирования, но и предоставляет полезные возможности, такие как "прозрачное протоколирование" и наследование автоматов [5].

1.3. Языковые средства

Для построения системы отладки требуется возможность получать информацию о структуре классов отлаживаемого приложения. Современными языками, предоставляющими набор инструментов для получения информации о структуре классов во время работы, являются *Java* [7] и *C#* [8]. Принадлежащие им наборы инструментов *java.lang.** и *Reflection* во многом аналогичны. Создание подобной системы для отладки автоматных программ было возможным на любом из этих языков, однако был выбран язык *C#*, как более новый и, потенциально, более мощный. Также в работе использовалась возможность этого языка, отсутствующая в языке *Java* — делегаты (*delegates*).

2. Библиотека *AFC*

2.1. Назначение

Назначение библиотеки *AFC* (*Automat Foundation Classes*) состоит в предоставлении программисту гибкого и простого в использовании набора инструментов для разработки автоматного приложения, позволяющего создать эффективную систему отладки, которая может избавить разработчика от необходимости внесения в программу строк, требуемых при разработке, но лишних при применении.

Под гибкостью в данном случае подразумевается возможность задания большого числа параметров работы автомата, оптимизирующих программу под конкретную задачу. Упомянутые параметры будут описаны в следующих разделах.

Под простотой использования подразумевается возможность быстро и с небольшими усилиями создать реализацию автомата, не указывая многие дополнительные параметры там, где это не представляется целесообразным.

В качестве примера части программы, необходимой на этапе разработки, но лишней на этапе использования, можно привести блок, отвечающий за протоколирование. На этапе разработки подробная запись всех действий, произведенных в автоматах и с автоматами, может подсказать место возможной ошибки и прояснить их работу. Однако во время применения программы файл протокола является не только не необходимым (поскольку пользователи редко интересуются устройством программы), но и лишним, поскольку подробный файл протокола за длительный промежуток времени может увеличиться до неприемлемо больших размеров.

2.2. Внутреннее устройство

Далее следует описание внутреннего устройства библиотеки. Пояснения относительно тех сущностей и возможностей, назначение которых не очевидно, будет дано в разд. 2.3.

2.2.1. Диспетчер автоматов

Класс *AutomatDispatcher*, реализованный в соответствии с паттерном *Singleton* [9], содержит в себе:

- ссылки на все зарегистрированные в нем автоматы — экземпляры класса *Automat* и его потомков;
- ссылки на «наблюдателей автоматов» — экземпляры классов, реализующих интерфейс *IAutomatWatcher*.

Диспетчер предоставляет возможность:

- получения списка зарегистрированных в нем автоматов;
- регистрации и разрегистрации в нем автоматов;
- добавления и удаления «наблюдателей автоматов»;
- получения автомата по имени;
- получения уникального имени автомата по предпочитаемому имени.

2.2.2. Автомат

Класс *Automat* реализует базовую функциональность автомата и интерфейс *IDisposable*. Он содержит:

- имя автомата;
- набор «вычислителей входных переменных» — экземпляры классов, реализующих интерфейс *InputVariableCalculator*;
- набор «исполнителей выходных воздействий» — экземпляры классов, реализующих интерфейс *IOutputActionPerformer*;
- набор «составителей протокола» — экземпляры классов, реализующих интерфейс *ILogger*;

- схему автомата — экземпляр класса *AutomatScheme*;
- контексты автомата и последнего вызванного события — произвольные объекты;
- текущее состояние — целое число;
- идентификатор способа вычисления входных переменных:
- *EveryTime* — вычислять при каждом обращении;
- *Cached* — запоминать значение, вычисленное при первом обращении после вызова последнего события;
- *PreCalculated* — вычислять и запоминать значения всех переменных, указанных в схеме, сразу при вызове события.
- закешированные значения вычисленных значений входных переменных;
- флаг «запущен ли автомат», то есть было ли передано автомату хотя бы одно событие.

Класс предоставляет возможность:

- вызова автомата с указанным событием и опционально указанным контекстом события;
- добавления и удаления «вычислителя входных переменных»
- добавления и удаления «исполнителя выходных воздействий»
- добавления и удаления «составителя протокола»;
- указания автомату, что его использование далее не предвидится.

2.2.3. ChangeEventRaiser

Класс *ChangeEventRaiser* реализует базовую функциональность сущности, сигнализирующей о своем изменении. Он предоставляет возможность:

- подписки на событие изменения экземпляра;
- временной блокировки посылки события.

2.2.4. Схема автомата

Класс *AutomatScheme* является потомком класса *ChangeEventRaiser* и реализует интерфейс *ICloneable*. Класс является представлением поведения автомата в виде метаданных. Он содержит:

- начальное состояние автомата — число;
- набор схем состояний — экземпляров класса *StateScheme*;
- набор схем событий — экземпляров класса *EventScheme*;
- набор схем входных переменных — экземпляров класса *VariableScheme*;
- набор схем выходных воздействий — экземпляров класса *ActionScheme*;
- набор схем переходов — экземпляров класса *TransitionScheme*.

Он предоставляет возможность:

- добавления, удаления и перечисления упомянутых схем компонент автомата;
- поиска одной из перечисленных схем по идентификатору;
- поиска всех схем переходов из заданного идентификатором состояния;
- создания собственной копии.

2.2.5. DescribedAutomatAttribute

Класс *DescribedAutomatAttribute* содержит данные, общие для всех компонент автомата — описание и идентификатор.

2.2.6. Массив чисел

Класс *IntArray* является потомком класса *ChangeEventRaiser* и реализует функциональность индексированного набора чисел, сигнализирующего о своем изменении. Он предоставляет возможность:

- получения и установки числа по индексу;
- добавления числа;
- удаления числа по значению и по индексу;
- получения количества чисел;

- преобразования в строку.

2.2.7. Схема состояния

Класс *StateScheme* является потомком класса *DescribedAutomatAttribute* и реализует интерфейс *ICloneable*. Является представлением состояния автомата. Содержит массив чисел — идентификаторы выходных воздействий, вызываемых при переходе в это состояние, и предоставляет возможность создания собственной копии.

2.2.8. Схема перехода

Класс *TransitionScheme* является потомком класса *DescribedAutomatAttribute* и реализует интерфейс *ICloneable*. Класс является представлением перехода автомата. Он содержит:

- набор исходных состояний — массив чисел;
- конечное состояние — целое число;
- набор выходных воздействий — массив чисел;
- условие перехода — строка;
- дерево условия — представление условия в виде бинарного дерева.

Он предоставляет возможность:

- получения списка используемых в условии событий и входных переменных;
- создания собственной копии.

2.2.10. Схемы остальных компонент автомата

Классы *VariableScheme*, *ActionScheme* и *EventScheme* являются потомками класса *DescribedAutomatAttribute* и реализуют интерфейс *ICloneable*. Они являются представлениями входных переменных, выходных воздействий и событий автомата соответственно и расширяют функциональность базового класса лишь возможностью создания собственных копий.

2.2.11. Вычислитель входной переменной

Интерфейс *InputVariableCalculator* обязывает класс реализовать следующие возможности:

- проверить предоставляет ли данный экземпляр возможность вычисления заданной входной переменной;
- вычислить заданную входную переменную.

2.2.12. Исполнитель выходного воздействия

Интерфейс *IOutputActionPerformer* обязывает класс реализовать следующие возможности:

- проверить позволяет ли данный экземпляр возможность исполнить заданное выходное воздействие;
- исполнить заданное выходное воздействие.

2.2.13. Составитель протокола

Интерфейс *ILogger* обязывает реализующий его класс предоставить возможности протоколирования следующих действий:

- изменение схемы автомата —экземпляра класса *AutomatScheme*;
- изменение начального состояния автомата;
- изменение режима вычисления входных переменных;
- создание автомата;
- завершение работы автомата;
- передача автомату события;
- начало вычисления входных переменных при режиме вычисления *PreCalculated*;
- завершение вычисления входных переменных при режиме вычисления *PreCalculated*;
- завершение обработки события;
- запрос значения входной переменной;

- возвращения значения входной переменной из списка кешированных значений при режиме вычисления *Cached*;
- возвращения значения входной переменной из списка кешированных значений при режиме вычисления *PreCalculated*;
- начало вычисления входной переменной;
- окончание вычисления входной переменной;
- начало выполнения выходного воздействия;
- окончание выполнения выходного воздействия;
- начало проверки условия перехода;
- завершение проверки условия перехода отрицательным результатом;
- начало выполнения перехода;
- окончание выполнения перехода;
- начало вхождения в состояние;
- добавление и удаление составителя протокола;
- добавление и удаление вычислителя входной переменной;
- добавление и удаление исполнителя выходного воздействия.

2.2.14. Простой автомат

Класс *SimpleAutomat* является потомком класса *Automat* и реализует интерфейсы *InputVariableCalculator* и *IOutputActionPerformer*. Он реализует поведение автомата, который сам вычисляет некоторые или все входные переменные и выполняет некоторые или все выходные воздействия.

2.3. Принципы работы

2.3.1. Схема

Как было сказано выше, схема автомата содержит в себе начальное состояние в числовом виде и наборы схем состояний, событий, входных переменных, выходных воздействий и переходов. Каждая из схем компонентов

автомата порождена от класса *DescribedAutomatAttribute*, и поэтому содержит идентификатор — число и описание — строку. Для схем состояний, событий, входных переменных и выходных воздействий идентификатор задает число, определяющее соответствующий компонент в классическом представлении — графе переходов. Таким образом, у схемы события *eI* идентификатор будет равен единице.

В классическом представлении автомата переходам не задаются идентификаторы, однако в данной работе было принято решение придать переходам обозначения вида *tI*, где число (единица) также соответствует значению идентификатора. Заметим, что обычно для переходов, в отличие от других компонентов автомата, сложно задать краткое словесное описание. В данной библиотеке описание перехода совпадает со строковым представлением условия, что позволяет выделить общего предка для схем всех компонентов автомата.

Экземпляр класса *AutomatScheme* непрерывно поддерживает свою корректность — в нем нет схем переходов, использующих несуществующие состояния, события, входные переменные и выходные воздействия. Кроме того, отсутствуют схемы состояний, при переходе в которые требуется вызывать несуществующие выходные воздействия. При выполнении изменений схемы автомата, которые могут привести к переходу схемы в некорректное состояние, происходит проверка целостности, и, в случае нахождения ссылок на несуществующие компоненты, происходит добавление безымянных схем компонентов — схем без описания. Также при переходе автомата в состояние, отсутствующее в схеме, вызове события, запросе на вычисление входной переменной или на выполнение выходного воздействия, отсутствующих в схеме, безымянные элементы добавляются в схему.

Также при каждом добавлении схемы компонента в схему автомата происходит подписка на событие об изменении компонента, а при удалении — отписка. Таким образом, при каком-либо изменении схемы компонента, добавленного в схему автомата, и при изменении начального состояния, не

являющегося компонентом и хранящегося в схеме автомата в виде числа, происходит вызов события, сигнализирующего о том, что схема автомата была изменена.

При добавлении в схему автомата схемы компонента, для которого уже есть однотипный элемент с таким же идентификатором, старый элемент замещается новым. Следовательно, при наличии безымянной схемы компонента, добавление именованной схемы компонента с таким же идентификатором является корректной операцией, оставляющей лишь именованную схему компонента.

2.3.2. Взаимодействие

Взаимодействие автомата с окружающей средой могут быть четырех типов [1]:

- вычисление входных переменных;
- выполнение выходных воздействий;
- обращения к состояниям других автоматов;
- передача другим автоматам событий.

В данной работе предлагается чуть более жесткий подход, не уменьшающий, однако, функциональности автоматного программирования. Проверку состояний других автоматов следует производить из блоков реализации входных переменных, а передачу другим автоматам событий — из блоков реализации выходных воздействий.

В случае, когда внешняя среда взаимодействует с одним автоматом, а он, в свою очередь, с другим, причем второй автомат с внешней средой не контактирует, имеет смысл инкапсулировать первый автомат во второй. Для этого предусмотрен контекст автомата, который может быть инициализирован каким-либо значением при создании автомата и передается в блоки реализации выходных воздействий и входных переменных. В нем предполагается хранение других автоматов, взаимодействующих с рассматриваемым автоматом, и прочих данных, необходимых для работы автомата.

Некоторые входные переменные и выходные воздействия могут быть скрыты от внешнего мира, являясь неотъемлемой частью автомата, и их можно реализовать в том же классе автомата (потомке класса *Automat*). Однако часть входных переменных и выходных воздействия взаимодействуют с внешним миром, и поэтому их целесообразно реализовывать в других частях программы. Также допустима ситуация, когда некоторые события и входные переменных должны быть доступны сразу нескольким автоматам. В этом случае предусмотрена возможность задания автомату нескольких «вычислителей входных переменных» и «исполнителей выходных воздействий». При запросе автомата на вычисление входной переменной (или выполнение выходного воздействия), происходит последовательное обращение ко всем «вычислителям входных переменных» (или «исполнителям выходных воздействий»), с целью выяснить, кто из них реализует необходимое действие. В случае нахождения такого «вычислителя» (или «исполнителя»), последовательные обращения прекращаются, и действие производится.

Также предусмотрена передача вместе с событием и контекста события, который затем передается в блоки реализации вычисления входных переменных и выходных воздействий. Это позволяет объединять однотипные события в группы, передавая вместе с событием дополнительную информацию.

2.3.3. Работа автомата

При создании автомат запрашивает у диспетчера уникальное имя на основе предпочитаемого — того, которое задал ему разработчик. Затем, установив себе полученное уникальное имя, автомат регистрируется в диспетчере. После этого автомат готов к работе.

При наличии схемы автомата, содержащей всю необходимую информацию о работе, отсутствует необходимость писать еще какой-либо код — реакция автомата на событие осуществляется в строгом соответствии со схемой. Однако программисту предоставляется возможность переопределить метод обработки события и написать собственный код, в таком случае задание схемы не

обязательно. При работе с использованием схемы проверка условий переходов происходит в следующем порядке: сначала проверяются групповые условия — те, у которых набор начальных состояний содержит более одного элемента, а затем проверяются остальные. Проверка и тех и других происходит в порядке возрастания идентификатора, являющегося числом.

Когда автомат получает уведомление о том, что он больше не будет использоваться, выполняется процедура разрегистрации у диспетчера автоматов, и, в случае, если более нет ни одной ссылки на автомат, то он может быть удален сборщиком мусора (*Garbage Collector*).

2.3.4. Протоколирование

При выполнении любого из элементарных действий, подлежащих протоколированию и перечисленных при описании «составителя протокола», происходит последовательный вызов соответствующих методов всех «составителей протокола». В случае работы автомата с использованием схемы, информация обо всех действиях будет запротоколирована. Однако при самостоятельной реализации реакции автомата на событие стоит либо вручную протоколировать действия, связанные с переходами, либо смириться с тем, что эти действия протоколироваться не будут. Отметим, что действие «начало вхождения в состояние» протоколируется только в случае работы автомата по схеме и когда состояние требует выполнения выходных воздействий при входе в него. Остальные же действия протоколируются в любом случае. При этом отметим, что в соответствующие методы «составителей протокола» передается вся необходимая информация о произошедшем действии.

2.3.5. Условия переходов

При создании новой схемы перехода, а также при изменении условия уже существующей, происходит создание представления условия в виде бинарного дерева. Корректной строкой условия считается булевская формула, состоящая из следующих элементов:

- круглых скобок;
- знака «&», обозначающего логическое «и»;
- знака «|», обозначающего логическое «или»;
- знака «!», обозначающего логическое отрицание;
- переменных вида *en* и *xn*, обозначающих соответственно событие и входную переменную с номером *n*.

2.3.6. Исключительные ситуации

При некорректной работе с библиотекой могут возникнуть исключительные ситуации. Однако для каждой из них предусмотрено свое исключение — класс, порожденный от *AFCEException*, в свою очередь, являющегося потомком класса *Exception*. Экземпляр соответствующего класса сигнализирует об ошибке. Перечислим исключительные ситуации и возникающие в них исключения:

- автомат, запись о котором отсутствует у диспетчера, пытается зарегистрироваться. Исключение *AutomatNotFoundException*;
- повторная регистрация автомата. Исключение *RepeatedAutomatRegistrationException*;
- передача *null* вместо автомата при регистрации или разрегистрации. Исключение *NullAutomatRegistrationException*;
- удаление «наблюдателя автоматов», запись о котором отсутствует у диспетчера. Исключение *WatcherNotFoundException*;
- регистрация либо разрегистрация автомата с неуникальным именем. Исключение *NotUniqueNameException*;
- запрос на получение автомата по имени, запись о котором отсутствует у диспетчера. Исключение *NameNotFoundException*;
- отсутствует реализация запрашиваемой входной переменной. Исключение *VariableNotImplementedException*;

- отсутствует реализация запрашиваемого выходного воздействия. Исключение *ActionNotImplementedException*;
- запрос на получение значения переменной, которое не было вычислено, в режиме вычисление *PreCalculated*. Исключение *VariableNotPreCalculatedException*;
- некорректная строка условия для перехода. Исключение *InvalidConditionException*;
- передача пустого набора в качестве исходных состояний для перехода. Исключение *InvalidYFromException*.

2.4. Использование библиотеки

В случае применения схемы автомата и отсутствия необходимости использовать контекст автомата, достаточно создать экземпляр класса *Automat*, передав ему имя, схему и наборы «составителей протокола», «вычислителей входных переменных» и «выходных воздействий».

«Составители протокола», «вычислители входных переменных» и «исполнители выходных воздействий» могут быть добавлены и после создания автомата, в том числе и после начала его работы. В таком случае блоки реализации входных переменных и выходных воздействий придется реализовывать вне автомата. Однако создание потомка класса *Automat* дает возможность инициализировать контекст автомата для дальнейшего применения. Когда необходимо некоторые или все входные переменные и выходные воздействия реализовать в самом автомате, следует породить свой класс от класса *Automat*, реализовать им интерфейсы *InputVariableCalculator* и *IOutputActionPerformer* и добавить автомату в качестве «вычислителя входной переменной» и «исполнителя выходного воздействия» самого себя. Также возможно порождение от класса *SimpleAutomat*, который сам реализует соответствующие интерфейсы и добавляет себя в соответствующие списки. В таком случае стоит лишь перекрыть соответствующие методы реализации вычисления входных переменных и выполнения выходных воздействий.

Использование схемы автомата позволяет сократить объем кода, определяющего реализацию автомата. Например, на рис. 1 тринадцать строк полностью задают поведение автомата с указанным начальным состоянием, четырьмя состояниями, тремя событиями, четырьмя входными переменными, десятью выходными воздействиями и двенадцатью переходами.

```
Scheme.StartY = 1;
Scheme.AddTransition(1, 1, "(e1|e2) &x1", 1,1);
Scheme.AddTransition(2, 1, "(e1|e2) &!x1&x2", 1,2);
Scheme.AddTransition(3, 1, "(e1|e2) &!x1&x3", 1,3);
Scheme.AddTransition(4, 1, "e3", 3,5,8);
Scheme.AddTransition(5, 1, "e2 &x1&!x2&!x3", 2);
Scheme.AddTransition(6, 2, "e1 &x1&!x2&!x3", 2, 7);
Scheme.AddTransition(7, 2, "(e1|e2) &(x1|x2|x3)", 1);
Scheme.AddTransition(8, 2, "e2 &x1&!x2&!x3", 2, 4);
Scheme.AddTransition(9, 3, "e2", 3, 4);
Scheme.AddTransition(10, 3, "e1", 4, 6, 9);
Scheme.AddTransition(11, 4, "!x4", 4, 4);
Scheme.AddTransition(12, 4, "x4", 1, 10);
```

Рис. 1. Пример кода, описывающего автомат

Первым передаваемым параметром является идентификатор перехода. В данном случае — это числа от 1 до 12. Затем идет номер начального состояния перехода (в случае группового перехода стоит указать массив чисел). После этого следует строковое представление условия перехода, а затем — номер конечного состояния. В конце, опционально, указываются выходные воздействия перехода.

Заметим, что подобный код, инициализирующий схему автомата, полностью изоморфен графу переходов автомата.

Созданные на основе предложенной библиотеки автоматы допускают изменение схемы во время работы, что корректно отразится на реакции автомата на передаваемые события.

Ввиду регистрации автоматов у диспетчера автоматов, ссылки на них хранятся, пока они не разрегистраются, и поэтому сборщик мусора (*Garbage Collector*) не может высвободить память, занимаемую теми автоматами, использование которых больше не предвидится. Для устранения возможности появления этой проблемы следует сигнализировать автомату об окончании его использования, когда надобность в нем исчезает.

Также отметим важный момент, необходимый для отладки приложения предложенной системой отладки — программист должен объявить точку входа отладочной системы в программу, как доступный другим классам (*public*) статический (*static*) метод. Обычно основной точкой входа является статический метод *Main*, не являющийся доступным другим классам. Его можно объявить таковым, добавив модификатор доступа *public*, и тогда он сможет стать точкой входа в программу для системы отладки. Однако программист может объявить и другой статический метод точкой входа в программу отладочной системы, сделав его доступным другим классам.

Для корректного использования библиотеки программисту нужно знать ее программный интерфейс и понимать назначения предоставляемых ею возможностей. Описание всех классов библиотеки и предоставляемых ими возможностей было приведено выше. Здесь же прокомментируем лишь часть интерфейса библиотеки, знание которого достаточно для создания автоматов на ее основе.

В большинстве случаев для задания логики переходов по состояниям автомата достаточно задать схему автомата. Программный интерфейс и реализация с комментариями класса *AutomatScheme* и компонентов автомата (*StateScheme*, *VariableScheme*, *ActionScheme*, *EventScheme* и *TransitionScheme*) приведены в приложении 1. В приложении 2 приведены интерфейс и код метода, который следует перекрыть в случае необходимости реализовать логику переходов автомата, не используя схему автомата.

Для реализации кода, отвечающего за вычисление входных переменных и выполнение выходных воздействий, программисту необходимо знать соответствующий программный интерфейс. Интерфейсы и комментарии к методам, позволяющим добавлять «вычислителей входных переменных» и «выполнителей выходных воздействий», приведены в приложении 4. В приложении 3 приведены интерфейсы «вычислителя» и «выполнителя» (*InputVariableCalculator* и *IOutputActionPerformer*). Все комментарии приведены в стиле, являющимся стандартом комментирования для языка C#.

Отличия библиотеки от предыдущей версии

В работе [5] автором была представлена библиотека, являющаяся предшественником данной. Их объединяет объектноориентированный подход к автоматному программированию, возможность наследования автоматов и возможность вести «прозрачное протоколирование». Отличием является то, что в данной библиотеке появилась возможность задавать принцип работы автомата, то есть логику его переходов, не только кодом, как в предыдущей библиотеке, но и посредством метаинформации, в данном случае схемы автомата. Это позволяет контролировать все параметры работы автомата в каждый момент времени и позволяет написать систему отладки.

3. Демонстрационное приложение

Для демонстрации возможностей автоматной системы отладки, описанной в разд. 4, требовалось написать приложение, на основе библиотеки, представленной в разд. 2. Отметим, что созданное приложение не претендует на оптимальность использованного алгоритма, а всего лишь демонстрирует некоторые (отнюдь не все) возможности написанной библиотеки и служит примером для работы отладочной системы.

3.1. Постановка задачи

Требуется написать приложение, демонстрирующее решение следующей задачи. Прямоугольная рабочая область, с вертикальными и горизонтальными границами, называемая *лабиринтом*, разбита на квадратные клетки горизонтальными и вертикальными линиями. Часть клеток является *стенами*. Известно, что клетки, прилегающие к границам *лабиринта*, являются *стенами*. Клетки, не являющиеся *стенами*, являются *свободными*. На некоторых *свободных* клетках располагаются *призы*. Одна из *свободных* клеток, на которой нет *приза*, является *стартовой*. В ней первоначально находится *путешественник*. Он может перемещаться из одной клетки в другую, если они обе являются *свободными* и имеют общую сторону. Известно, что, перемещаясь указанным способом, он может попасть в любую *свободную* клетку *лабиринта*. Оказавшись в клетке с *призом*, *путешественник* может его взять. При этом *приз* в клетке исчезает, а *путешественник* считается несущим *приз*. Придя с *призом* в *стартовую* клетку, *путешественник* может *приз* положить. При этом *путешественник* перестает считаться несущим *приз*. Одновременно он может нести не более одного *приза*. При отображении следует указывать направление, в котором смотрит *путешественник*. Направление может быть лишь одним из следующих - вверх, вниз, влево, вправо. Начальное направление произвольно. *Путешественник* может передвинуться в другую клетку, только если направление движения

совпадает с направлением, в котором он смотрит. Путешественник может изменять направление, поворачиваясь направо и налево под прямым углом. Путешественник не знает общего количества призов и может узнавать, *свободна* ли клетка, только если она спереди, слева или справа от него, относительно направления его движения. Требуется отнести все *призы* в *стартовую* клетку и заявить об окончании работы.

3.2. Словесное описание предложенного алгоритма

Путешественник, ищущий *приз*, движется вперед (в ту сторону, которая совпадает с его направлением движения), пока у него есть такая возможность. Если такой возможности нет, но с одной из сторон (справа или слева относительно направления движения) расположена *свободная* клетка — путешественник поворачивается в эту сторону. Если свободно с обеих сторон — он поворачивает налево.

Если движение вперед невозможно, и клетки слева и справа являются *стенами*, он возвращается по пройденному пути, пока либо не обнаружит спереди, слева или справа *свободную* клетку, либо не окажется в *стартовой клетке*. Если, возвращаясь, он обнаружил слева, справа или спереди *свободную* клетку, то он продолжает поиск приза.

Если, возвращаясь, он окажется в *стартовой* клетке, из которой движения вперед и в стороны также невозможно, он считает работу выполненной. Оказавшись в клетке с *призом*, он берет его, возвращается в *стартовую* клетку, затем возвращается в клетку нахождения *приза* и продолжает поиск. Эту задачу целесообразно решать, используя автоматный подход.

3.3. Автомат A1 («Путешественник»)

3.3.1. Словесное описание

Автомат, представленный классом *Walker*, реализует поведение *путешественника*. Получая в качестве события информацию о том, в какого рода клетке он находится — в *свободной*, в клетке с *призом* или в *стартовой* клетке, он выполняет соответствующие действия.

Интерес представляет процедура возвращения с *призом* в *стартовую* клетку и последующего возвращения в клетку нахождения *приза*. При движении в поисках *приза*, формируется стек, на вершину которого при каждом перемещении помещается направление движения. Таким образом, «запоминается» дорога, по которой шел *путешественник*.

Из выходных воздействий наибольший интерес представляет воздействие z_4 , передающее событие e_4 автомату A2 («Компас»), которое отвечает за возвращение назад по пройденному пути. При возвращении, направление шага выбирается противоположным направлению, находящемуся на вершине стека, которое затем из него удаляется. Следовательно, существует возможность запомнить пройденный путь и возвратиться по нему.

В момент взятия *приза* копируется текущий стек, который назовем исходным, и формируется стек возврата — стек, возврат по которому из *стартовой* клетки приведет в клетку нахождения *приза*. Он формируется из исходного, путем перечисления в обратном порядке направлений, противоположных направлениям из исходного стека. После возвращения в *стартовую* клетку с *призом*, вместо текущего стека, подставляется стек возврата, по которому и происходит возвращение в клетку нахождения *приза*. Затем восстанавливается исходный стек и продолжается работа.

3.3.2. Схема связей

Схема связей автомата *A1* («Путешественник») приведена на рис. 2.

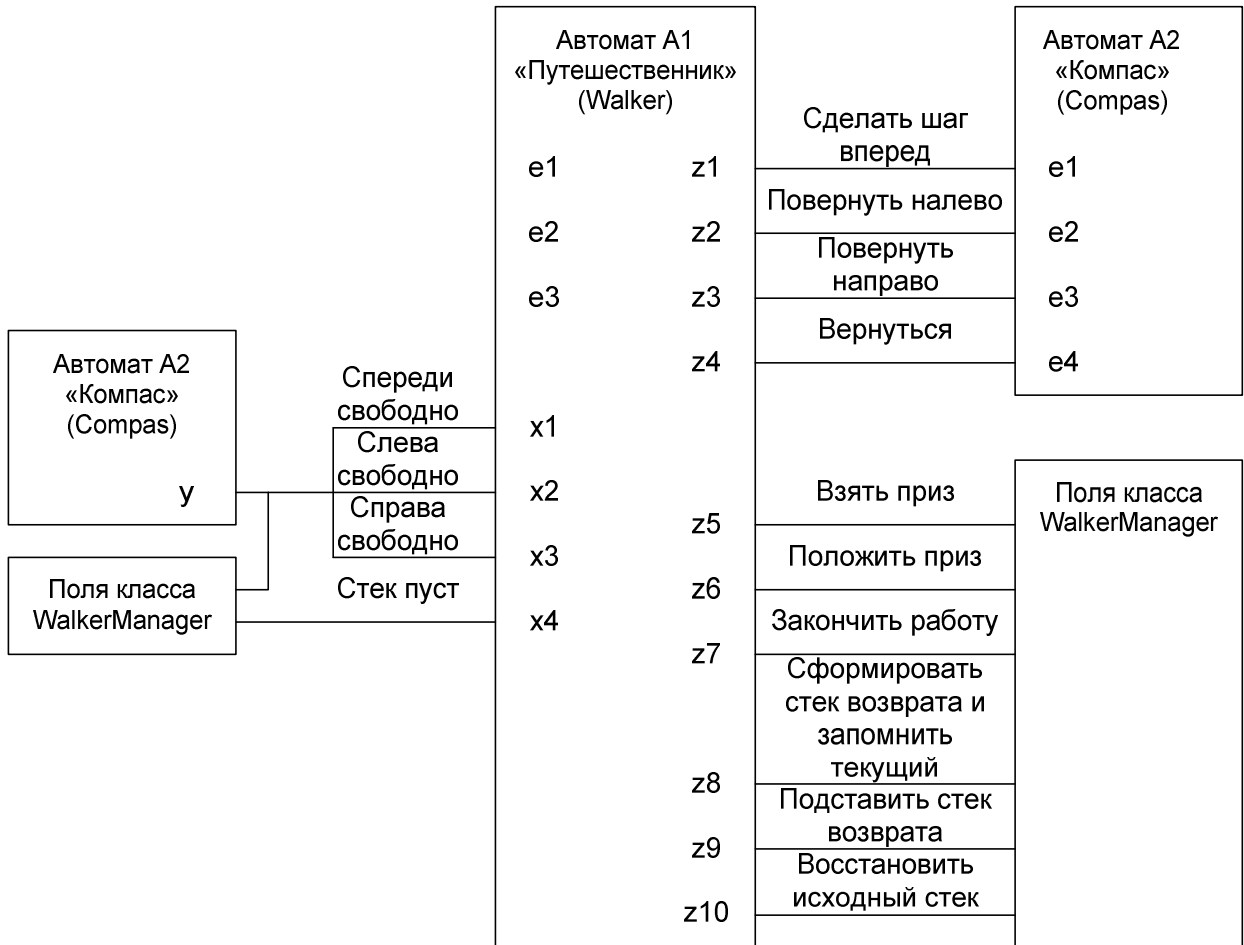
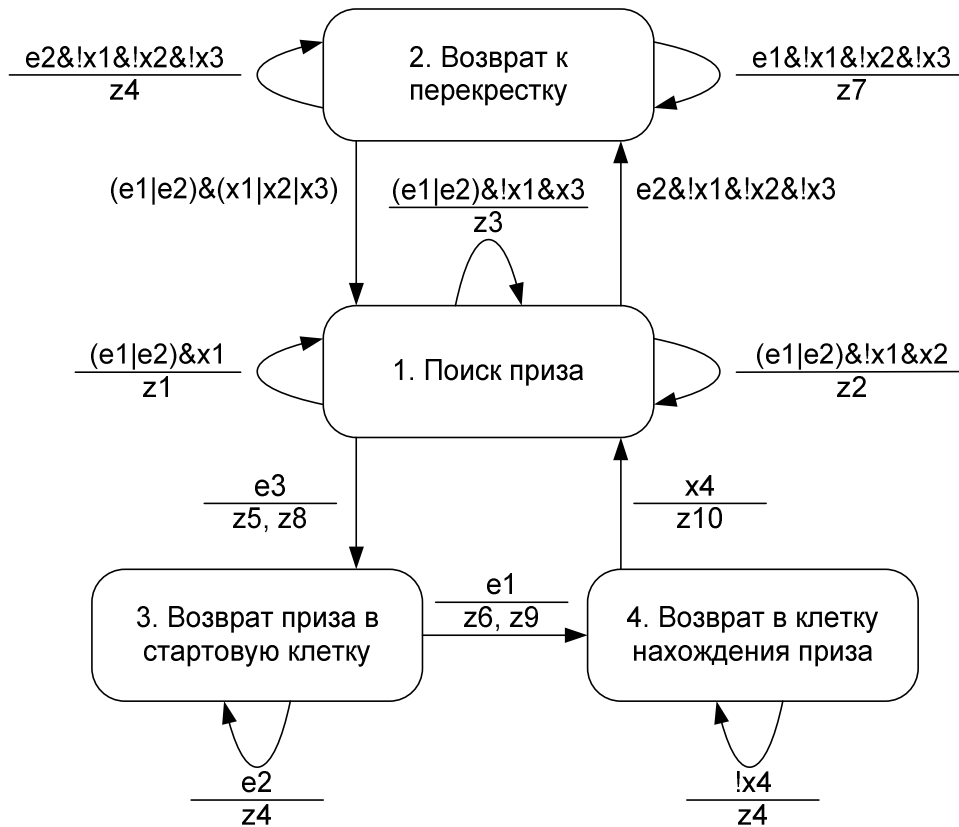


Рис. 2. Схема связей автомата *A1* («Путешественник»)

3.3.3. Граф переходов

Граф переходов автомата $A1$ («Путешественник») представлен на рис. 3.



e1	Оказался в стартовой клетке
e2	Оказался в свободной клетке
e3	Оказался в клетке с призом

x1	Спереди свободно
x2	Слева свободно
x3	Справа свободно
x4	Стек пуст

z1	Сделать шаг вперед
z2	Повернуть налево
z3	Повернуть направо
z4	Вернуться
z5	Взять приз
z6	Положить приз
z7	Закончить работу
z8	Сформировать стек возврата и запомнить текущий
z9	Подставить стек возврата
z10	Восстановить исходный стек

Рис. 3. Граф переходов автомата $A1$ («Путешественник»)

3.4. Автомат A2 («Компас»)

3.4.1. Словесное описание

Автомат, представленный классом *Compas*, реализует функциональность компаса. Его состояние определяет направление, в котором смотрит *путешественник*. Получая события, требующие повернуться, он переходит в соответствующее состояние. Получив событие, требующее движение вперед, он совершает соответствующие выходные воздействия — выполняет шаг в том направлении, в котором смотрит *путешественник*, и добавляет на вершину стека направление сделанного шага. Получив событие, требующее возвращения назад, автомат «поворачивается», если текущее направление не соответствует направлению возвращения, либо совершает шаг, в направлении, обратном направлению на вершине стека, и удаляет верхний элемент стека.

3.4.2. Схема связей

Схема связей автомата A2 («Компас») представлена на рис. 4.

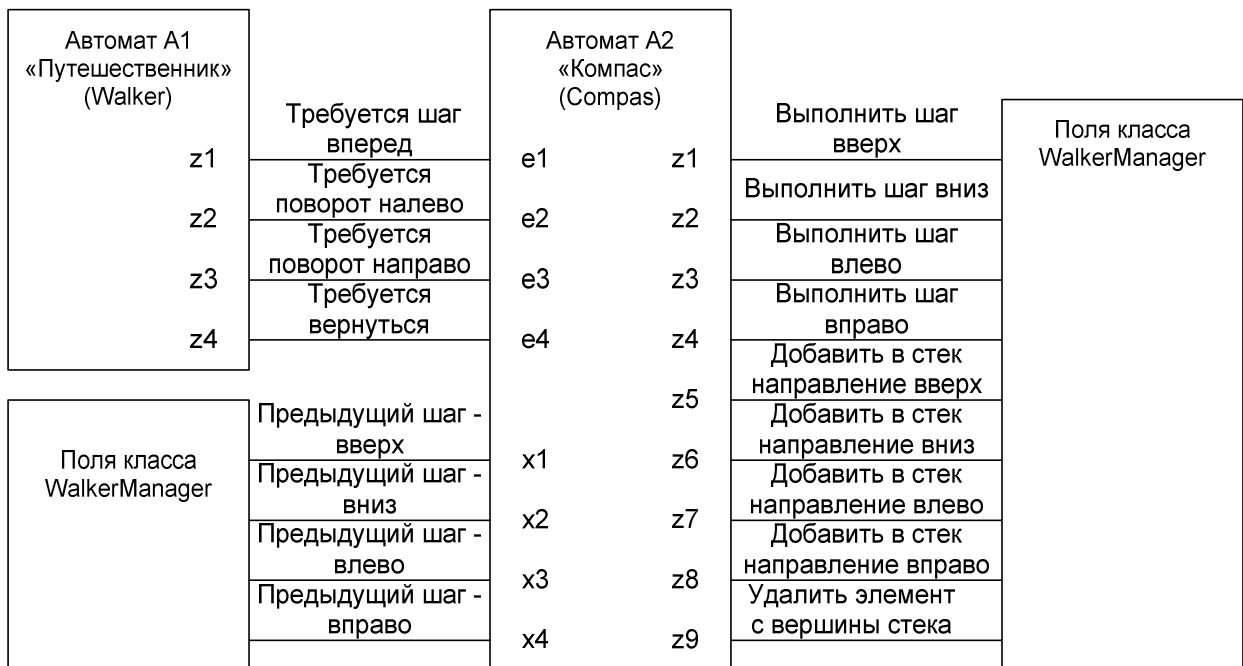
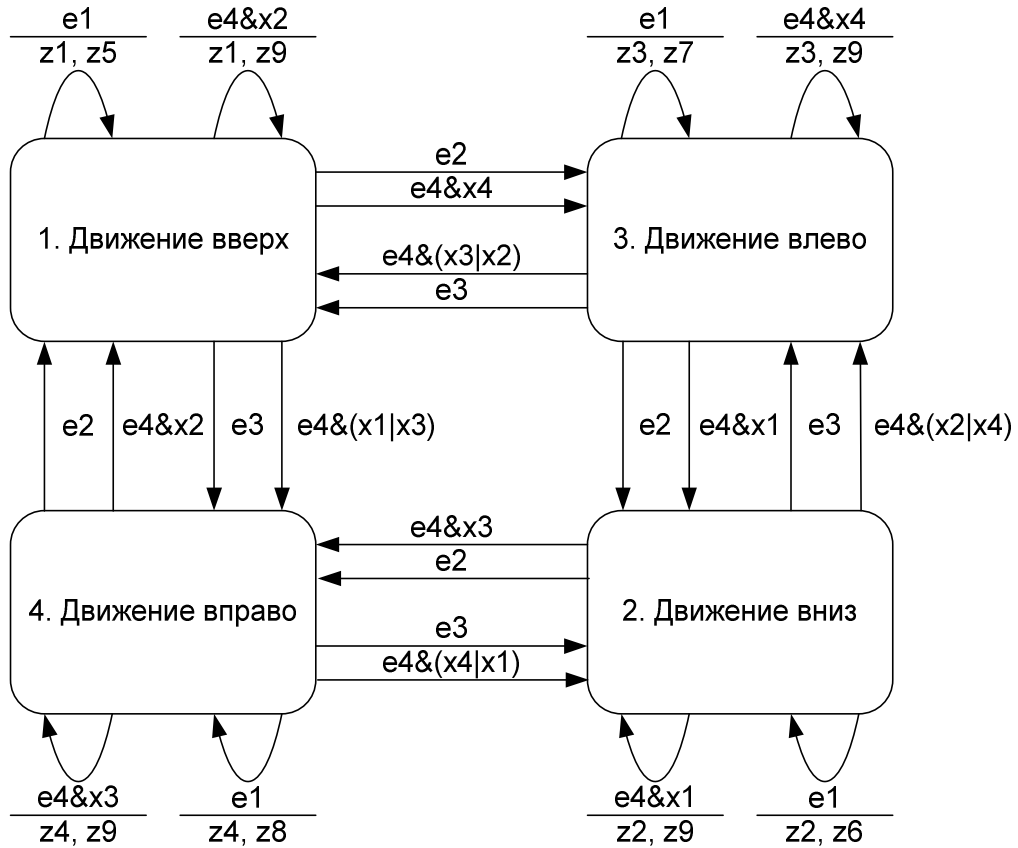


Рис. 4. Схема связей автомата A2 («Компас»)

3.4.3. Граф переходов

Граф переходов автомата *A2* («Компас») представлен на рис. 5.



e1	Требуется шаг вперед
e2	Требуется поворот налево
e3	Требуется поворот направо
e4	Требуется вернуться

x1	Предыдущий шаг - вверх
x2	Предыдущий шаг вниз
x3	Предыдущий шаг влево
x4	Предыдущий шаг - вправо

z1	Выполнить шаг вверх
z2	Выполнить шаг вниз
z3	Выполнить шаг влево
z4	Выполнить шаг вправо
z5	Добавить в стек направление вверх
z6	Добавить в стек направление вниз
z7	Добавить в стек направление влево
z8	Добавить в стек направление вправо
z9	Удалить элемент с вершины стека

Рис. 5. Граф переходов автомата *A2* («Компас»)

3.5. Работа приложения

При запуске приложения создается экземпляр класса *WalkerManager*, который, в свою очередь, создает экземпляры классов *Walker* — автомат *A1* («Путешественник») и *Compas* — автомат *A2* («Компас») и реализует

вычисление их входных переменных и выходных воздействие. Содержание *лабиринта* хранится в двумерном массиве, а стеки — в одномерных массивах. При необходимости сделать очередной шаг, автомату *AI* («*Путешественник*») передается событие, указывающее на то, в клетке какого рода оказался *путешественник*. Выходные воздействия автоматов изменяют поля класса *WalkerManager*, значения которых используются при отображении.

3.6. Внешний вид приложения

На рис. 6 изображен внешний вид приложения.

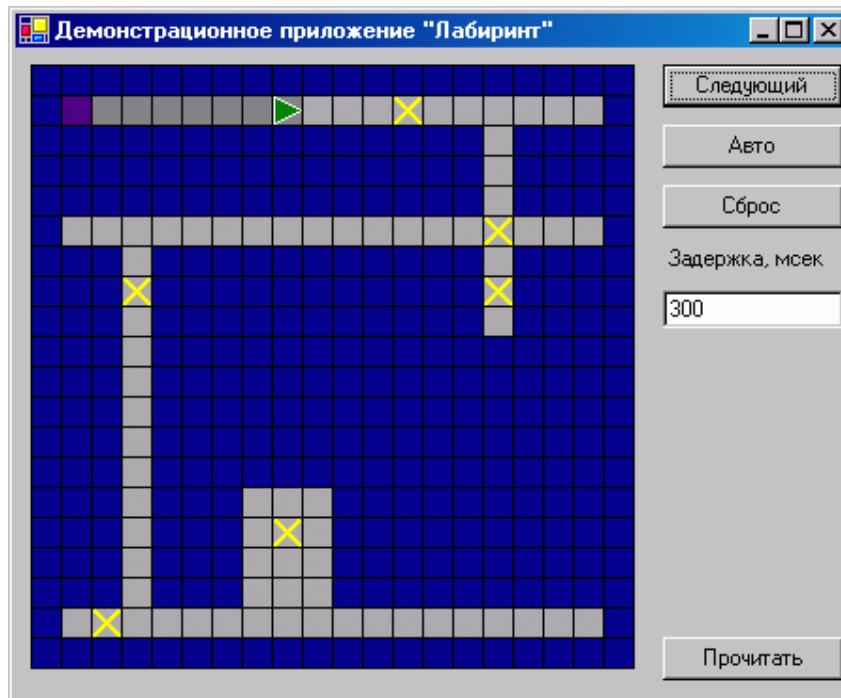


Рис. 6. Демонстрационное приложение «Лабиринт». Этап поиска приза

Лабиринт показан в виде клеток, разделенных черными линиями. Стены изображены синими квадратами. Светло-серыми квадратами отмечены свободные клетки, где путешественник еще не побывал. Темно-серыми квадратами представлены свободные клетки, где путешественник уже был. Фиолетовый квадрат обозначает стартовую клетку. Желтые крестики — призы. Темно-зеленый треугольник обозначает путешественника, направление которого указывает угол треугольника, повернутый в направлении, параллельном одной из сторон лабиринта. Путешественник, взявший приз, отображается светло-зеленым треугольником, как показано на рис. 7.

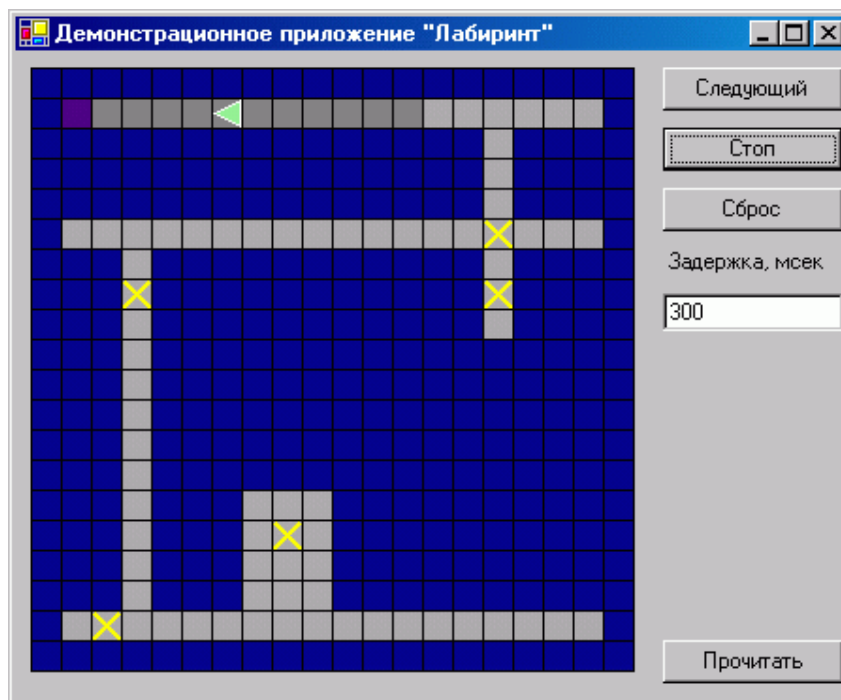


Рис. 7. Демонстрационное приложение «Лабиринт». Этап возвращения приза

В начале приложение находится в *ручном* режиме. В этом режиме события не передаются автоматам до тех пор, пока не нажата кнопка **Следующий**. При нажатии этой кнопки автомату *AI* («Путешественник») передается очередное событие. При нажатии кнопки **Авто** приложение переходит в *автоматический* режим и начинает передавать события указанному автомату через промежутки времени, длина которых в миллисекундах задается в окне **Задержка, мсек**. При этом надпись на кнопке **Авто** изменяется на **Стоп**.

При нажатии на кнопку **Стоп** приложение возвращается в *ручной* режим, в котором оно находилось изначально, и надпись на этой кнопке обратно изменяется на **Авто**.

Нажатие кнопки **Сброс** восстанавливает исходное положение, направление и состояние *путешественника*, восстанавливает начальное положение *призов* и помечает все свободные клетки как не посещенные.

Кнопка **Прочитать** позволяет загрузить конфигурацию задачи — расположение *стен*, *свободных* клеток, *стартовой* клетки и *призов* из файла.

4. Автоматная система отладки

4.1. Принципы работы

Один из наборов инструментов платформы *.NET Framework*, а именно *Reflection*, позволяет получать доступ к структуре классов скомпилированной программы, загрузив ее как сборку (*Assembly*).

Система отладки открывает указанное приложение как сборку и перечисляет все статические (*static*) методы классов в ней, которые доступны другим классам (*public*). Если среди них есть метод обычного входа в программу *Main*, то тогда он сразу назначается точкой входа в программу системы отладки. Пользователю также предоставляется возможность указать любой метод из перечисленных. Если метод описан, как получающий в параметре массив строк, пользователь может ввести строковые параметры для его запуска. Далее экземпляр основного класса системы отладки добавляется (как «наблюдатель автоматов») в список диспетчеру автоматов. Затем экземпляр основного класса системы отладки запускает указанное приложение в созданном для него отдельном потоке (*thread*), в котором вызывается метод, назначенный, как точка входа в программу системы отладки (с указанными параметрами, если метод допускает их передачу).

Так как диспетчер автоматов реализован с использованием паттерна *Singleton*, то автоматы отлаживаемого приложения регистрируются у того же экземпляра класса *AutomatDispatcher*, у которого в списке «наблюдателей автоматов» уже находится система отладки.

Таким образом, предлагаемая система получает уведомления о регистрации каждого автомата из отлаживаемого приложения. Получив очередной автомат, система отладки добавляет экземпляр основного класса в список «составителей протокола» автомата, и таким образом получает уведомления о каждом действии, происходящем с ним. При необходимости остановить работу приложения,

система отладки приостанавливает главный поток — тот, в котором был вызван метод, назначенный точкой входа системы отладки в приложение.

В случае, когда отлаживаемое приложение имеет однопоточную архитектуру, его работа полностью остановится. Для многопоточного приложения работа остальных потоков продолжится, однако пока они не используют автоматы, их работа не имеет отношения к автоматной системе отладки. Когда же один из потоков приложения, главный поток которого приостановлен, производит какое-либо действие с автоматом, то управление переходит к системе отладки, как к одному из «составителей протокола».

Однако те методы системы отладки, которые вызываются при каких-либо действиях с автоматом, написаны с использованием оператора *lock* — два потока не могут одновременно выполнять их, и поток, вызвавший один из этих методов вторым, вынужден ждать, пока первый поток не выйдет из защищенной области.

Система отладки написана таким образом, что главный поток отлаживаемого приложения приостанавливается в момент выполнения одного из этих методов. Таким образом, после остановки главного потока ни одного действия с автоматами не произойдет, пока он не продолжит работу. Это показывает, что разработанная система отладки работоспособна даже для многопоточного отлаживаемого приложения.

4.2. Структура системы отладки

4.2.1. Основной класс системы отладки

Основным классом, экземпляр которого выполняет работу системы отладки, является класс *ADS*. Он предоставляет возможность подписки на следующие события:

- загружена сборка;
- отлаживаемое приложение запущено;
- отлаживаемое приложение приостановлено;

- работа отлаживаемого приложения возобновлена;
- список автоматов, оставшихся после завершения работы приложения, очищен;
- приложения завершило работу;
- автомат зарегистрировался;
- автомат разрегистрировался;
- состояние автомата изменилось;
- схема автомата изменилась;
- добавлена либо удалена точка остановки;
- изменился статус автомата либо его компонента;
- требуется внести строку в протокол;

Класс содержит:

- ссылку на диспетчера автоматов;
- ссылку на сборку;
- список статических методов отлаживаемого приложения, доступных другим классам;
- ссылку на метод, выбранный точкой входа в отлаживаемое приложение системы отладки;
- строковые параметры для вызова метода, выбранного точкой входа в отлаживаемое приложение системы отладки;
- ссылку на главный поток приложения;
- флаг «сборка загружена»;
- флаг «отлаживаемое приложение запущено»;
- флаг «отлаживаемое приложение приостановлено»;
- список зарегистрированных автоматов;
- набор автоматов и их компонент, на которые установлены точки остановки;
- набор статусов автоматов и их частей;
- флаг «необходимо остановиться, когда указанный либо какой-либо автомат не совершит очередное действие»;

- указатель на автомат, при очередном действии которого следует приостановить приложение;
- флаги, указывающие, следует ли заносить в протокол действия связанные с состояниями, событиями, входными переменными, выходными воздействиями и переходами автоматов;
- сдвиг следующей строки протокола.

Он предоставляет следующие возможности:

- загружать отлаживаемое приложение;
- запускать отлаживаемое приложение;
- приостанавливать отлаживаемое приложение;
- возобновлять работу приостановленного отлаживаемого приложения;
- прерывать работу отлаживаемого приложения;
- устанавливать и снимать точку остановки;
- возобновлять работу приостановленного приложения до любого действия любого автомата;
- возобновлять работу приостановленного приложения до любого события указанного автомата;
- очищать список автоматов, оставшихся после завершения работы приложения.

4.2.2. Классы, реализующие интерфейс системы отладки

Основная форма приложения является экземпляром класса *MainForm*. Форма, позволяющая выбирать метод, являющийся точкой входа в приложения системы отладки, реализована классом *ParamsForm*. Формы, выводящие информацию об автоматах, состояниях, событиях, входных переменных, выходных воздействиях и переходах реализованы классами соответственно *AutomatsForm*, *StatesForm*, *EventsForm*, *VarForm*, *InfForm* и *TranForm*, которые являются потомками класса *OwnedForm*. Все формы хранят ссылку на экземпляр класса *ADS* и взаимодействуют с ним, вызывая методы, обращаясь к свойствам и подписываясь на события.

4.3. Работа системы отладки

4.3.1. Открытие приложения, ввод параметров, запуск, остановка, прерывание

При открытии система отладки загружает указанную сборку. Если возникает ошибка, то система сигнализирует об этом посредством исключения *LoadAssemblyErrorException*, порожденного, как и другие исключения системы отладки, от класса *ADSEException*. При удачной загрузке формируется список доступных статических методов. Если он содержит метод *Main*, то этот метод сразу назначается главным — точкой входа в приложение системы отладки. Далее возможно задание параметров вызова главного метода — набора строк. Запуск осуществляется посредством создания потока, называемого главным, в котором вызывает главный метод. Если этот метод отсутствует, то система сигнализирует об ошибке исключением *StartErrorException*. При вызове метода определяются требуемые параметры. В случае, когда метод действительно требует для запуска массив строк, ему передается упомянутый массив, который по умолчанию является пустым. Если метод не получает параметров, то вызов осуществляется без них. Если же параметры вызова не являются строковыми, то в качестве параметров передаются пустые ссылки (*null*). По завершению работы приложения система сигнализирует об этом. При возникновении исключения в отлаживаемой программе, оно передается в параметрах события, сигнализирующего об окончании работы.

При необходимости остановить работу приложения останавливается главный поток. Как было объяснено выше, это останавливает любую работу с автоматами, даже если приложения создало другие потоки. Возобновление работы приложения происходит при возобновлении работы главного потока. Прерывание работы приложения возобновляет работу главного потока, если он остановлен, и прерывает его выполнение. Обо всех упомянутых действиях система сигнализирует соответствующими событиями.

4.3.2. Наблюдение за автоматами

При регистрации очередного автомата система добавляет экземпляр основного класса в список «составителей протокола» для него и благодаря этому получает уведомления обо всех элементарных действиях, происходящих с автоматом, со всей необходимой информацией о произошедшем действии. Это позволяет системе отладки хранить у себя корректную информацию об автоматах и их компонентах, называемую статусом. Статус автомата или его компонента состоит из строки, содержащей информацию о происходящем или произошедшем действии, и идентификатора, указывающего на тип действия. Также для входных переменных хранится последнее вычисленное значение.

Если действие начало происходить, или происходит, то тип действия является *Active*. Когда действие заканчивается, тип устанавливается равным *Finished*, и остается таковым до любого следующего действия, затем становясь *Passive*. Система своевременно сигнализирует об изменении статуса автомата или его компонента, что позволяет интерфейсной части корректно отображать информацию. Отображение текущего статуса автоматов и их компонентов аналогично отображению наблюдаемых выражений в стандартных средствах отладки.

Также система протоколирует те действия с автоматами, в которых заинтересован программист. Протоколирование ведется структурно, что улучшает понимание текста протокола и позволяет рассматривать его, как стек вызовов — один из элементов стандартных средств отладки.

4.3.3. Точки останова

Система позволяет устанавливать на автомат или его компонент точку останова. Когда происходит действие с автоматом или его компонентом, на который установлена точка останова, система приостанавливает работу приложения, предварительно сообщив о произошедшем действии. Точки

остановки в данной системе аналогичны точкам остановки в стандартных системах отладки.

4.3.4. Работа с автоматами по шагам

В то время как приложение остановлено, система позволяет не только возобновить его работу в прежнем режиме, но и продолжить работу до первого действия, произошедшего либо с выбранным автоматом, либо с любым автоматом из отлаживаемого приложения. Это происходит путем проверки, того не был ли затребован подобный режим работы во время изменения статуса автомата или его компонента. В случае положительного результата, после сигнализации об изменении статуса автомата или его компонента, система приостанавливает работу приложения. Возможность работы с автоматами по шагам аналогична возможности работы с приложением по шагам, предоставляемой стандартными средствами отладки.

4.4. Внешний вид системы отладки

На рис. 8 показан внешний вид системы отладки в момент запуска.

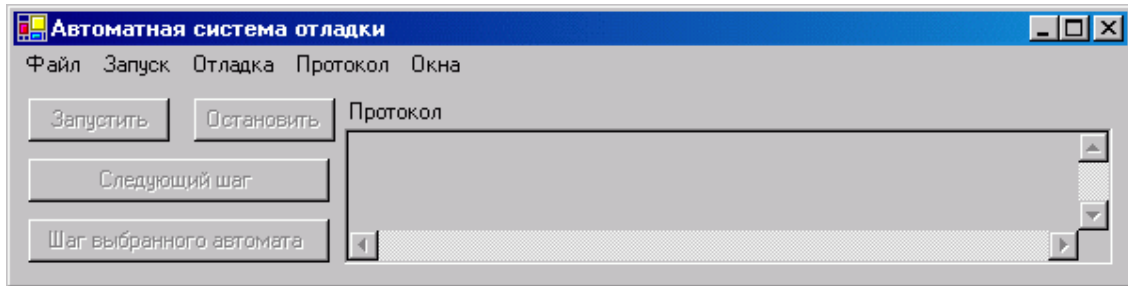


Рис. 8. Автоматная система отладки

В окне **Протокол** отображается структурированный протокол работы автоматов, состоящий из записей о действиях, произошедших с интересующими программиста компонентами автоматов, а также информация об основных действиях системы, таких, как загрузка сборки или завершение работы приложения. Пример структурированного протокола можно увидеть на рис. 9.

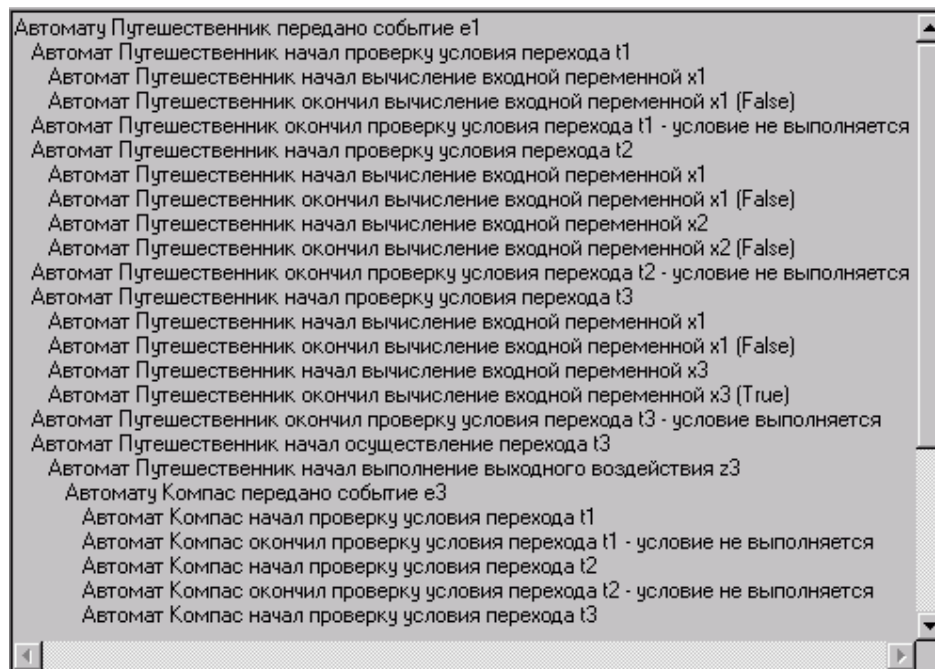


Рис. 9. Пример структурированного протокола

При завершении работы отлаживаемого приложения исключением, в окне **Протокол** отображается информация о произошедшей ошибке, содержащаяся в исключении.

Пункт меню **Файл** содержит подпункты **Открыть** и **Выход**, предназначенные, соответственно, для открытия отлаживаемого приложения и выхода из системы отладки, возможного лишь когда отлаживаемое приложение не запущено.

Пункт меню **Запуск** содержит подпункты **Запустить**, **Параметры** и **Удалить автоматы**, предназначенные, соответственно, для запуска отлаживаемого приложения, выбора параметров (точки входа и параметров запуска) и удаления автоматов, оставшихся после завершения работы приложения. Подпункт **Запустить** продублирован соответствующей кнопкой на рабочей области окна приложения.

Пункт меню **Отладка** содержит подпункты **Остановить**, **Возобновить**, **Прервать**, **Следующий шаг** и **Шаг выбранного автомата**. Предназначены они для следующих целей:

- **Остановить** - приостанавливает работу приложения;
- **Возобновить** - возобновляет работу приостановленного приложения;
- **Прервать** - прерывает работу запущенного приложения;
- **Следующий шаг** - возобновляет работу приложения до первого действия, произошедшего с любым из автоматов;
- **Шаг выбранного автомата** - возобновляет работу приложения до первого действия, произошедшего с выбранным автоматом.

Подпункты **Остановить**, **Следующий шаг** и **Шаг выбранного автомата** продублированы соответствующими кнопками на рабочей области окна приложения.

Заметим, что подпункт **Запустить** эквивалентен подпункту **Возобновить** во время работы приложения.

Пункт меню **Протокол** содержит подпункты **Состояния**, **События**, **Входные переменные**, **Выходные воздействия** и **Переходы**, указывающие протоколировать ли действия, происходящие с соответствующими компонентами автоматов.

Пункт меню **Окна** содержит подпункты **Автоматы**, **Состояния**, **События**, **Входные переменные**, **Выходные воздействия**, **Переходы**, указывающие отражать ли окна, показывающие информацию о соответствующих сущностях.

На рис. 10 в качестве примера показан вид окна, отображающего информацию о двух автоматах. Первый из них находится в четвертом состоянии, а второй – в первом.

Имя	Тип	Состояние	Статус
Компас	MazeApp.Compas	y4 - Движение вправо	Обработано e3
Путешественник	MazeApp.Walker	y1 - Поиск приза	Обработано e1

Рис. 10. Информация об автоматах

Каждая строка представляет один автомат и отображает имя автомата, имя класса, идентификатор и описание состояния, и статус, содержащий информацию о последнем действии, произошедшем с автоматом. В этом и во всех других окнах, отображающих автоматы либо их компоненты, зеленый цвет означает, что действие начало происходить, синий — что действие закончилось. Жирный шрифт соответствует тому, что на автомат или его компонент установлена точка остановки.

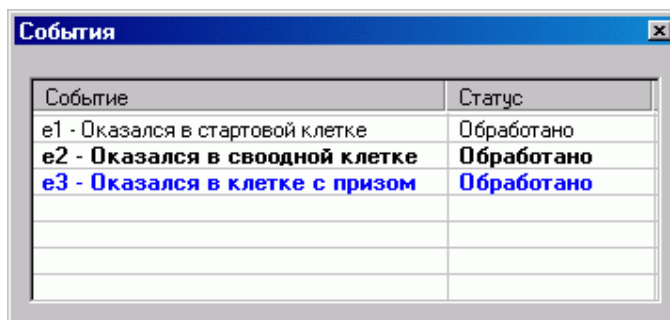
На рис. 11 показан вид окна, отображающего информацию о состояниях автомата, выбранного в окне автоматов.

Состояние	Выходные воздействия	Статус
y1 - Поиск приза		Вход осуществлен
y2 - Возврат к перекрестку		
y3 - Возврат приза в стартовую клетку		Вход осуществлен
y4 - Возврат в клетку нахождения приза		

Рис. 11. Информация о состояниях выбранного автомата

Каждая строка соответствует одному состоянию и содержит идентификатор и описание, список выходных воздействий и статус.

На рис. 12 показан вид окна, отображающего информацию о событиях выбранного автомата.

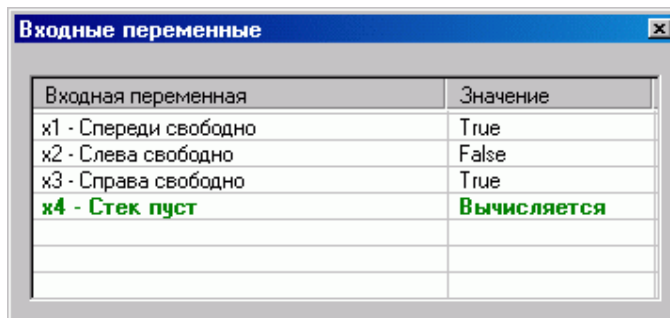


Событие	Статус
e1 - Оказался в стартовой клетке	Обработано
e2 - Оказался в своодной клетке	Обработано
e3 - Оказался в клетке с призом	Обработано

Рис. 12. Информация о событиях

Каждая строка представляет событие и отображает идентификатор и описание события, и статус, содержащий информацию о последнем действии, произошедшем с этим событием.

На рис. 13 показан внешний вид окна, отображающего информацию о входных переменных выбранного автомата.



Входная переменная	Значение
x1 - Спереди свободно	True
x2 - Слева свободно	False
x3 - Справа свободно	True
x4 - Стек пуст	Вычисляется

Рис. 13. Информация о входных переменных

Каждая строка представляет одну входную переменную и отображает идентификатор и описание входной переменной и статус — Вычисляется, True, False (два последних – вычисленное значение).

На рис. 14 показан внешний вид окна, отображающего информацию о выходных воздействиях выбранного автомата.

Выходное воздействие	Статус
z1 - Сделать шаг вперед	Выполнено
z2 - Повернуть налево	
z3 - Повернуть направо	Выполнено
z4 - Вернуться	Выполнено
z5 - Взять приз	Выполнено
z6 - Положить приз	Выполнено
z7 - Закончить работу	
z8 - Сформировать стек возврата и зап...	Выполнено
z9 - Подставить стек возврата	Выполнено
z10 - Восстановить исходный стек	Выполнено

Рис. 14. Информация о выходных воздействиях

Каждая строка представляет одно выходное воздействие и отображает идентификатор и описание выходного воздействия и статус — выполняется ли данное выходное воздействие.

На рис. 15 показан внешний вид окна, отображающего информацию о переходах выбранного автомата.

Т	Начальные состояния	Условие	Конечное состояние	Выходные воздействия	Статус
t1	y1	(e1 e2)&x1	y1 - Поиск приза	z1	Не выполняется
t2	y1	(e1 e2)&!x1&x2	y1 - Поиск приза	z2	Не выполняется
t3	y1	(e1 e2)&!x1&x3	y1 - Поиск приза	z3	Не выполняется
t4	y1	e3	y3 - Возврат приза ...	z5, z8	Осуществлен
t5	y1	e2&!x1&!x2&!x3	y2 - Возврат к пере...		
t6	y2	e1&!x1&!x2&!x3	y2 - Возврат к пере...	z7	
t7	y2	(e1 e2)&(x1 x2 x3)	y1 - Поиск приза		
t8	y2	e2&!x1&!x2&!x3	y2 - Возврат к пере...	z4	
t9	y3	e2	y3 - Возврат приза ...	z4	Не выполняется
t10	y3	e1	y4 - Возврат в клет...	z6, z9	Осуществлен
t11	y4	!x4	y4 - Возврат в клет...	z4	Не выполняется
t12	y4	x4	y1 - Поиск приза	z10	Осуществляется

Отображаемые переходы

Все переходы
 Из текущего состояния
 Из выбранного состояния

Рис. 15. Информация о переходах

Каждая строка представляет один переход и отображает идентификатор, начальные состояния, условие, идентификатор и описание конечного состояния, выходные воздействия и статус — проверяется ли условие перехода, либо результат проверки, либо осуществляется ли переход. Допускается возможность выбора отображаемых переходов — всех, либо из текущего состояния, либо из состояния, выбранного в окне состояний.

4.5. Пример отладки

Рассмотрим пример работы отладочной системы. Предположим, что допущена ошибка где-то в отладочном приложении. В данном случае ошибка заменена кодом `throw new Exception();`, генерирующим исключительную ситуацию. Во время работы приложения ошибка выглядит так, как показано на рис. 16.

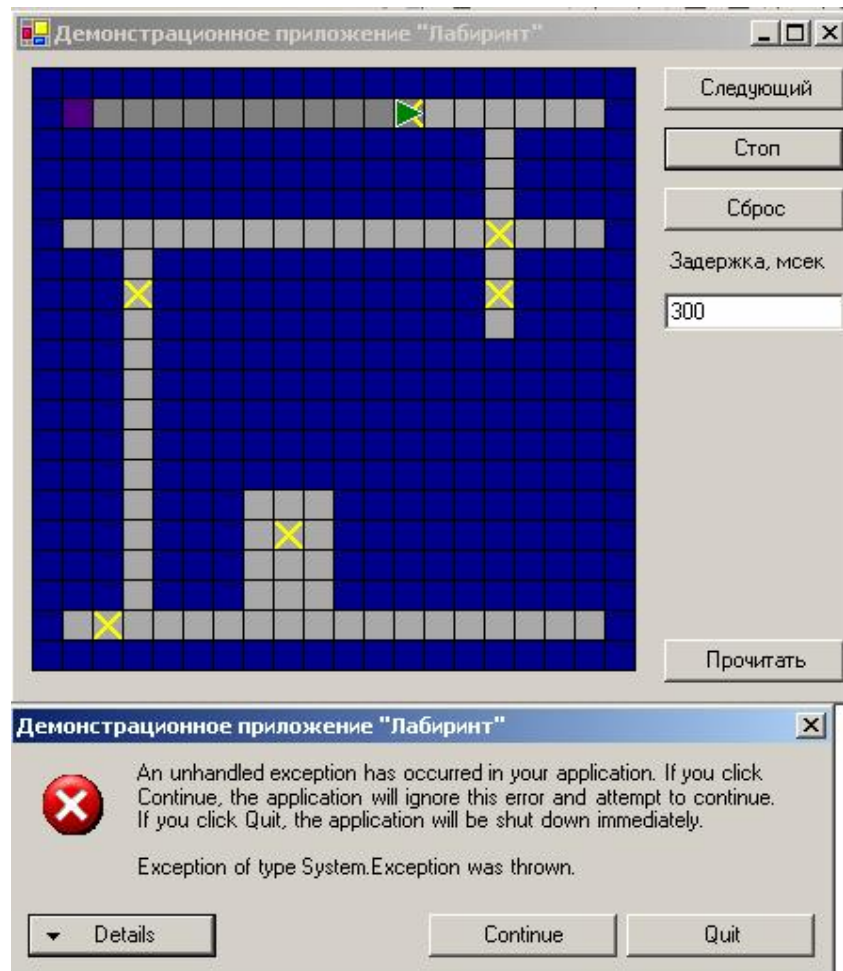


Рис. 16 Ошибка при работе приложения

Сообщение об ошибке довольно часто содержит достаточно информации о том, на какой строчке кода была сформирована исключительная ситуация. Здесь текст сообщения сознательно опущен, так как целью работы являлось создание системы отладки, позволяющей абстрагироваться от терминов объектно-ориентированного программирования и позволяющей мыслить автоматными

терминами. Если открыть приложение из системы отладки и дождаться появления исключительной ситуации, то можно увидеть картину, приведенную на рис. 17.

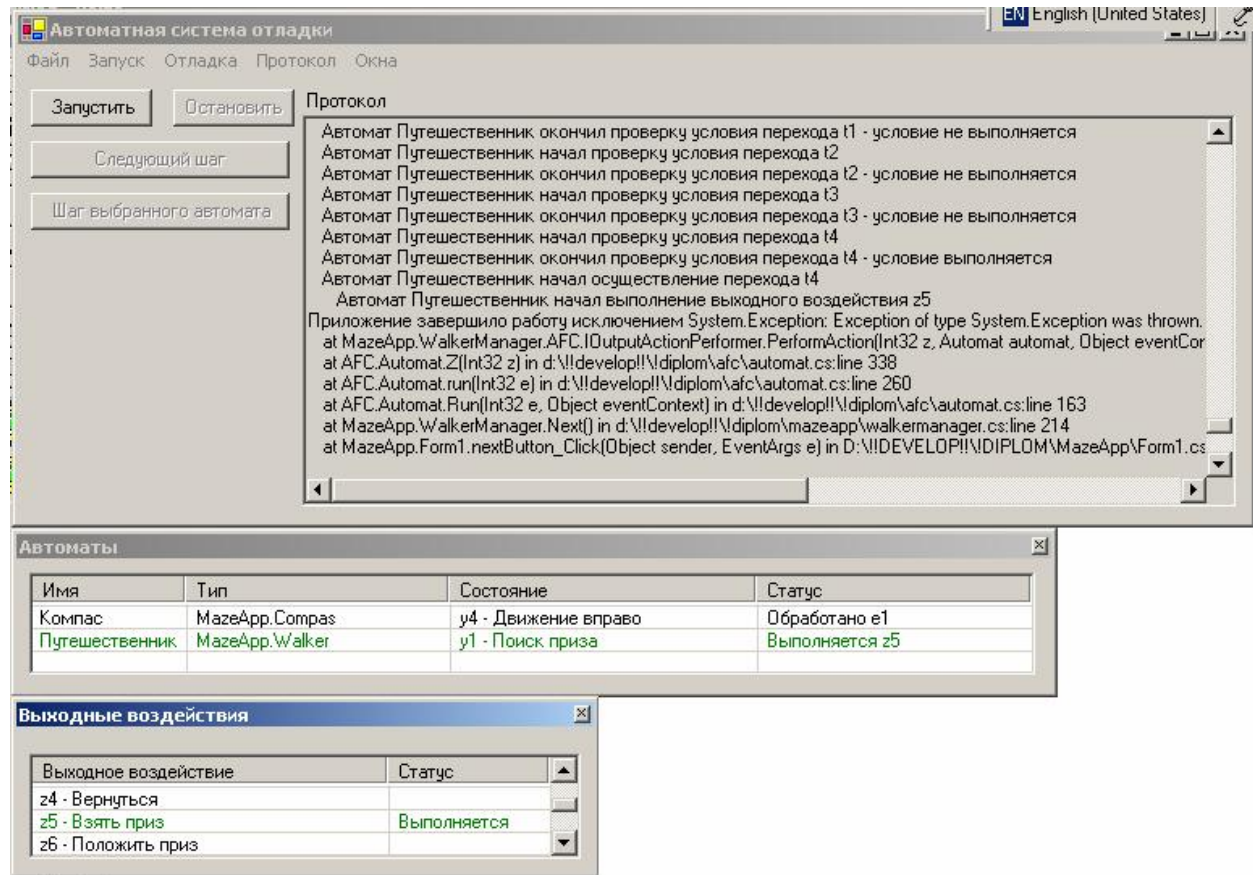


Рис. 17. Система отладки во время работы

Как следует из протокола, находящегося на главном окне, озаглавленном «Автоматная система отладки», приложение завершило работу исключительной ситуацией после начала выполнения автоматом Путешественник выходного воздействия z5. Это также следует из информации в окне «Автоматы», где указано, что статус автомата Путешественник — «Выполняется z5». Из окна «Выходные воздействия» становится ясно, что ошибка действительно возникла в коде, выполняющем выходное воздействие z5 «Взять приз».

Таким образом, система отладки автоматных приложений позволила локализовать ошибку с наименьшими трудозатратами — пришлось всего лишь

дождаться появления исключительной ситуации и проанализировать предоставляемую системой информацию.

ЗАКЛЮЧЕНИЕ

В представленной работе обоснована актуальность создания средства отладки, позволяющего абстрагироваться от способа реализации автоматов и сосредоточиться на автоматной логике. Проведен анализ существующих средств отладки, и построена система, предоставляющая инструменты для отладки автоматных программ, аналогичных наиболее используемым стандартным средствам. Перечислим предоставляемые возможности системы отладки:

- остановка работы отлаживаемого приложения в любой момент времени с возможностью последующего возобновления работы;
- остановка работы отлаживаемого приложения при выполнении действий с отмеченными компонентами автоматов и автоматами;
- отображение информации об автоматах и их компонентах;
- продолжение работы остановленного приложения до следующего действия, произошедшего с указанным либо произвольным автоматом;
- отображение структурированной последовательности произведенных действий;

Отлаживаемое приложение должно строиться на основе разработанной библиотеки, обладающей большинством возможностей аналогичных исследованных библиотек. Перечислим предоставляемые возможности разработанной библиотеки:

- гибкая настройка поведения автомата;
- простая реализация автоматов, не требующих расширенной настройки поведения;
- прозрачное протоколирование;
- наследование автоматов;
- динамическая компоновка автоматов из составляющих частей;
- изменение логики работы автомата во время работы программы;

- передача с событием дополнительной информации, позволяющей объединить однотипные события в группы;
- хранение в автомате дополнительной информации, позволяющей установить связи между автоматами;
- разнесение реализаций разных входных переменных и выходных воздействий по разным классам;
- протоколирование сразу несколькими способами;
- создание системы отладки для автоматов, построенных на основе данной библиотеки.

Создано приложение на основе предложенной библиотеки, на примере которого продемонстрированы возможности разработанной системы отладки. Приложение является примером использования предложенной библиотеки и демонстрирует простоту разработки автоматных программ.

Таким образом, применение предложенной библиотеки и системы отладки позволяет разрабатывать автоматные приложения, используя автоматы в проектной документации, исходном коде, протоколе и при отладке, что приводит к большему единообразию процесса работы и результата.

Список литературы

1. *Шалыто А.А.* Технология автоматного программирования //Мир ПК. 2003. № 10. <http://is.ifmo.ru>, раздел “Статьи”.
2. *Роббинс Дж.* Отладка приложений для Microsoft .NET и Microsoft Windows. М.: Русская редакция, 2004.
3. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5. <http://is.ifmo.ru>, раздел “Статьи”.
4. *Шалыто А.А., Туккель Н.И.* Танки и автоматы // ВУТЕ/Россия. 2003. № 2. <http://is.ifmo.ru>, раздел “Статьи”.
5. *Фельдман П.И., Шалыто А.А.* Совместное использование объектного и автоматного подходов в программировании. СПб.: СПбГУ ИТМО, 2004. <http://is.ifmo.ru>, раздел “Проекты”.
6. *Шалыто А.А., Наумов Л.А.* Реализация автоматов в объектно-ориентированных программах. <http://is.ifmo.ru>, раздел “Статьи”.
7. *Эккель Б.* Философия Java. СПб.: Питер, 2003.
8. *Либерти Дж.* Программирование на С#. М.: Символ-Плюс, 2003.
9. *Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

Приложение 1. Классы схемы автомата и схем компонентов

автомата

Класс AutomateScheme

```
using System;
using System.Collections;

namespace AFC
{
    /// <summary>
    /// Схема автомата
    /// </summary>
    public class AutomateScheme : ChangeEventRaiser, ICloneable
    {
        #region Private Members
        private SortedList m_States;
        private SortedList m_Events;
        private SortedList m_Variables;
        private SortedList m_Actions;
        private SortedList m_Transitions;
        private int m_StartY;
        #endregion

        #region Public properties
        /// <summary>
        /// Идентификатор начального состояния
        /// </summary>
        public int StartY
        {
            get
            {
                return m_StartY;
            }
            set
            {
                SuppressEvents(true);
                m_StartY = value;
                if (!m_States.ContainsKey(StartY))
                    AddState(StartY, "Без имени");
                Changed();
                SuppressEvents(false);
            }
        }

        /// <summary>
        /// Набор схем состояний - экземпляров класса StateScheme
        /// </summary>
        public ICollection States
        {
            get
            {
                return m_States.Values;
            }
        }
    }
}
```

```

/// <summary>
/// Набор схем событий - экземпляров класса EventScheme
/// </summary>
public ICollection Events
{
    get
    {
        return m_Events.Values;
    }
}

/// <summary>
/// Набор схем входных переменных - экземпляров класса VariableScheme
/// </summary>
public ICollection Variables
{
    get
    {
        return m_Variables.Values;
    }
}

/// <summary>
/// Набор схем выходных воздействий - экземпляров класса ActionScheme
/// </summary>
public ICollection Actions
{
    get
    {
        return m_Actions.Values;
    }
}

/// <summary>
/// Набор схем переходов - экземпляров класса TransitionScheme
/// </summary>
public ICollection Transitions
{
    get
    {
        return m_Transitions.Values;
    }
}

#endregion

#region Protected methods
protected void MembersChanged(ChangeEventRaiser sender)
{
    Changed();
    if (sender is StateScheme)
        CheckState(sender as StateScheme);
    if (sender is TransitionScheme)
        CheckTransition(sender as TransitionScheme);
}

protected void CheckState(StateScheme state)
{
    foreach(int z in state.Actions)
    {
        if (!m_Actions.ContainsKey(z))
            AddAction(z, "Без имени");
    }
}

```



```

protected void CheckTransition(TransitionScheme transition)
{
    foreach(int z in transition.Actions)
    {
        if (!m_Actions.ContainsKey(z))
            AddAction(z, "Без имени");
    }
    foreach(int y in transition.YFrom)
    {
        if (!m_States.ContainsKey(y))
            AddState(y, "Без имени");
    }
    if (!m_States.ContainsKey(transition.YTo))
        AddState(transition.YTo, "Без имени");
    foreach(int e in transition.UsedEvents)
    {
        if (!m_Events.ContainsKey(e))
            AddEvent(e, "Без имени");
    }
    foreach(int x in transition.UsedVariables)
    {
        if (!m_Variables.ContainsKey(x))
            AddVariable(x, "Без имени");
    }
}
#endregion

#region Public methods
/// <summary>
/// Добавляет схему состояния
/// </summary>
/// <param name="state">Схема состояния</param>
/// <returns>Схема состояния</returns>
public StateScheme AddState(StateScheme state)
{
    SuppressEvents(true);
    state.OnChanged += new
ObjectChangedEventHandler(this.MembersChanged);
    m_States[state.Value] = state;
    CheckState(state);
    Changed();
    SuppressEvents(false);
    return state;
}

/// <summary>
/// Удаляет схему состояния
/// </summary>
/// <param name="y">Идентификатор состояния</param>
public void RemoveState(int y)
{
    if (m_States.ContainsKey(y))
    {
        SuppressEvents(true);
        m_States.Remove(y);
        foreach(TransitionScheme transition in Transitions)
            CheckTransition(transition);
        Changed();
        SuppressEvents(false);
    }
}
}

```

```

/// <summary>
/// Удаляет схему состояния
/// </summary>
/// <param name="state">Схема состояния</param>
public void RemoveState(StateScheme state)
{
    RemoveState(state.Value);
}
/// <summary>
/// Добавляет схему состояния
/// </summary>
/// <param name="y">Идентификатор состояния</param>
/// <param name="description">Описание состояния</param>
/// <param name="inf">Набор идентификаторов выходных воздействий,
/// выполняемых при переходе в это состояние</param>
/// <returns>Схема состояния</returns>
public StateScheme AddState(int y, string description, params int[] inf)
{
    return AddState(new StateScheme(y, description, inf));
}

/// <summary>
/// Ищет схему состояния
/// </summary>
/// <param name="y">Идентификатор состояния</param>
/// <returns>Схема состояния</returns>
public StateScheme FindState(int y)
{
    return m_States[y] as StateScheme;
}

/// <summary>
/// Добавляет схему события
/// </summary>
/// <param name="ev">Схема события</param>
/// <returns>Схема события</returns>
public EventScheme AddEvent(EventScheme ev)
{
    ev.OnChanged += new ObjectChangedEventHandler(this.MembersChanged);
    m_Events[ev.Value] = ev;
    Changed();
    return ev;
}

/// <summary>
/// Удаляет схему события
/// </summary>
/// <param name="e">Идентификатор события</param>
public void RemoveEvent(int e)
{
    if (m_Events.ContainsKey(e))
    {
        SuppressEvents(true);
        m_Events.Remove(e);
        foreach(TransitionScheme transition in Transitions)
        {
            CheckTransition(transition);
            Changed();
            SuppressEvents(false);
        }
    }
}

/// <summary>
/// Удаляет схему события
/// </summary>

```

```

/// <param name="ev">Схема события</param>
public void RemoveEvent(EventScheme ev)
{
    RemoveEvent(ev.Value);
}

/// <summary>
/// Добавляет схему события
/// </summary>
/// <param name="e">Идентификатор события</param>
/// <param name="description">Схема события</param>
/// <returns></returns>
public EventScheme AddEvent(int e, string description)
{
    return AddEvent(new EventScheme(e, description));
}

/// <summary>
/// Ищет схему события
/// </summary>
/// <param name="e">Идентификатор события</param>
/// <returns>Схема события</returns>
public EventScheme FindEvent(int e)
{
    return m_Events[e] as EventScheme;
}

/// <summary>
/// Добавляет схему входной переменной
/// </summary>
/// <param name="var">Схема входной переменной</param>
/// <returns>Схема входной переменной</returns>
public VariableScheme AddVariable(VariableScheme var)
{
    var.OnChanged += new
    ObjectChangedEventHandler(this.MembersChanged);
    m_Variables[var.Value] = var;
    Changed();
    return var;
}

/// <summary>
/// Удаляет схему входной переменной
/// </summary>
/// <param name="x">Идентификатор входной переменной</param>
public void RemoveVariable(int x)
{
    if (m_Variables.ContainsKey(x))
    {
        SuppressEvents(true);
        m_Variables.Remove(x);
        foreach(TransitionScheme transition in Transitions)
            CheckTransition(transition);
        Changed();
        SuppressEvents(false);
    }
}

/// <summary>
/// Удаляет схему входной переменной
/// </summary>
/// <param name="var">Схема входной переменной</param>
public void RemoveVariable(VariableScheme var)
{

```

```

        RemoveVariable(var.Value);
    }

    /// <summary>
    /// Добавляет схему входной переменной
    /// </summary>
    /// <param name="x">Идентификатор входной переменной</param>
    /// <param name="description">Описание входной переменной</param>
    /// <returns>Схема входной переменной</returns>
    public VariableScheme AddVariable(int x, string description)
    {
        return AddVariable(new VariableScheme(x, description));
    }

    /// <summary>
    /// Ищет схему входной переменной
    /// </summary>
    /// <param name="x">Идентификатор входной переменной</param>
    /// <returns>Схема входной переменной</returns>
    public VariableScheme FindVariable(int x)
    {
        return m_Variables[x] as VariableScheme;
    }

    /// <summary>
    /// Добавляет схемы выходного воздействия
    /// </summary>
    /// <param name="inf">Схема выходного воздействия</param>
    /// <returns>Схема выходного воздействия</returns>
    public ActionScheme AddAction(ActionScheme inf)
    {
        inf.OnChanged += new
            ObjectChangedEventHandler(this.MembersChanged);
        m_Actions[inf.Value] = inf;
        Changed();
        return inf;
    }

    /// <summary>
    /// Удаляет схему выходного воздействия
    /// </summary>
    /// <param name="z">Идентификатор выходного воздействия</param>
    public void RemoveAction(int z)
    {
        if (m_Actions.ContainsKey(z))
        {
            SuppressEvents(true);
            m_Actions.Remove(z);
            foreach(TransitionScheme transition in Transitions)
                CheckTransition(transition);
            foreach(StateScheme state in States)
                CheckState(state);
            Changed();
            SuppressEvents(false);
        }
    }

    /// <summary>
    /// Удаляет схему выходного воздействия
    /// </summary>
    /// <param name="inf">Схема выходного воздействия</param>
    public void RemoveAction(ActionScheme inf)
    {
        RemoveAction(inf.Value);
    }

```

```

}

/// <summary>
/// Добавляет схему выходного воздействия
/// </summary>
/// <param name="z">Идентификатор выходного воздействия</param>
/// <param name="description">Описание выходного воздействия</param>
/// <returns></returns>
public ActionScheme AddAction(int z, string description)
{
    return AddAction(new ActionScheme(z, description));
}

/// <summary>
/// Ищет схему выходного воздействия
/// </summary>
/// <param name="z">Идентификатор выходного воздействия</param>
/// <returns>Схема выходного воздействия</returns>
public ActionScheme FindAction(int z)
{
    return m_Actions[z] as ActionScheme;
}

/// <summary>
/// Добавляет схему перехода
/// </summary>
/// <param name="transition">Схема перехода</param>
/// <returns>Схема перехода</returns>
public TransitionScheme AddTransition(TransitionScheme transition)
{
    SuppressEvents(true);
    transition.OnChanged += new
    ObjectChangedEventHandler(this.MembersChanged);
    m_Transitions[transition.Value] = transition;
    CheckTransition(transition);
    Changed();
    SuppressEvents(false);
    return transition;
}

/// <summary>
/// Удаляет схему перехода
/// </summary>
/// <param name="t">Идентификатор перехода</param>
public void RemoveTransition(int t)
{
    if (m_Transitions.ContainsKey(t))
    {
        m_Transitions.Remove(t);
        Changed();
    }
}

/// <summary>
/// Удаляет схему перехода
/// </summary>
/// <param name="transition">Схема перехода</param>
public void RemoveTransition(TransitionScheme transition)
{
    RemoveTransition(transition.Value);
}

/// <summary>
/// Добавляет схему группового перехода

```

```

/// </summary>
/// <param name="t">Идентификатор перехода</param>
/// <param name="yfrom">Набор идентификаторов начальных
/// состояний</param>
/// <param name="condition">Строковое представление условия</param>
/// <param name="yto">Конечное состояние</param>
/// <param name="inf">Набор идентификаторов выходных воздействий
/// выполняемых
/// при этом переходе</param>
/// <returns>Схема перехода</returns>
public TransitionScheme AddTransition(int t, int[] yfrom, string
condition, int yto, params int[] inf)
{
    return AddTransition(new TransitionScheme(t, yfrom, condition,
yto, inf));
}

/// <summary>
/// Добавляет схему перехода
/// </summary>
/// <param name="t">Идентификатор перехода</param>
/// <param name="yfrom">Идентификатор начального состояние</param>
/// <param name="condition">Строковое представление условия</param>
/// <param name="yto">Идентификатор конечного состояния</param>
/// <param name="inf">Набор идентификаторов выходных
/// воздействий</param>
/// <returns>Схема перехода</returns>
public TransitionScheme AddTransition(int t, int yfrom, string
condition, int yto, params int[] inf)
{
    return AddTransition(new TransitionScheme(t, yfrom, condition,
yto, inf));
}

/// <summary>
/// Ищет схему перехода
/// </summary>
/// <param name="t">Идентификатор перехода</param>
/// <returns>Схема перехода</returns>
public TransitionScheme FindTransition(int t)
{
    return m_Transitions[t] as TransitionScheme;
}

/// <summary>
/// Ищет схемы переходов из состояния
/// </summary>
/// <param name="y">Идентификатор состояния</param>
/// <returns>Набор схем переходов - экземпляров класса
/// TransitionScheme</returns>
public ICollection FindTransitionsFromY(int y)
{
    ArrayList singletr = new ArrayList();
    ArrayList grouptr = new ArrayList();
    foreach(TransitionScheme transition in m_Transitions.Values)
    {
        if (transition.YFrom.Contains(y))
        {
            if (transition.IsGroupTransition)
                grouptr.Add(transition);
            else
                singletr.Add(transition);
        }
    }
}

```

```

        grouptr.AddRange(singletr);
        return grouptr;
    }

    /// <summary>
    /// Создает эквивалентную себе схему
    /// </summary>
    /// <returns>Созданная схема</returns>
    public object Clone()
    {
        AutomatScheme sch = new AutomatScheme(StartY);
        foreach(ActionScheme inf in Actions)
            sch.AddAction(inf.Clone() as ActionScheme);
        foreach(VariableScheme var in Variables)
            sch.AddVariable(var.Clone() as VariableScheme);
        foreach(StateScheme state in States)
            sch.AddState(state.Clone() as StateScheme);
        foreach(EventScheme ev in Events)
            sch.AddEvent(ev.Clone() as EventScheme);
        foreach(TransitionScheme transition in Transitions)
            sch.AddTransition(transition.Clone() as TransitionScheme);
        return sch;
    }
#endregion

#region Constructor
    /// <summary>
    /// Создает схему автомата
    /// </summary>
    /// <param name="startY">Идентификатор начального состояния</param>
    public AutomatScheme(int startY)
    {
        m_States = new SortedList();
        m_Events = new SortedList();
        m_Variables = new SortedList();
        m_Actions = new SortedList();
        m_Transitions = new SortedList();
        StartY = startY;
    }
#endregion
}
}
}

```

Класс ActionScheme

```

using System;

namespace AFC
{
    /// <summary>
    /// Схема выходного воздействия
    /// </summary>
    public class ActionScheme : DescribedAutomatAttribute, ICloneable
    {
        #region Constructor
        /// <summary>
        /// Создает схему выходного воздействия
        /// </summary>
        /// <param name="z">Идентификатор выходного воздействия</param>
        /// <param name="description">Описание выходного воздействия</param>
        public ActionScheme(int z, string description)
            : base(z, description)
        {
        }
    }
}

```

```

    {
    }
    #endregion

    #region Public methods
    /// <summary>
    /// Создает эквивалентную себе схему
    /// </summary>
    /// <returns>Созданная схема</returns>
    public object Clone()
    {
        return new ActionScheme(Value, Description);
    }
    #endregion
}
}

```

Класс EventScheme

```

using System;

namespace AFC
{
    /// <summary>
    /// Схема события
    /// </summary>
    public class EventScheme : DescribedAutomatAttribute, ICloneable
    {
        #region Constructor
        /// <summary>
        /// Создает схему события
        /// </summary>
        /// <param name="e">Идентификатор события</param>
        /// <param name="description">Описание события</param>
        public EventScheme(int e, string description)
            : base(e, description)
        {
        }
        #endregion

        #region Public methods
        /// <summary>
        /// Создает эквивалентную себе схему
        /// </summary>
        /// <returns>Созданная схема</returns>
        public object Clone()
        {
            return new EventScheme(Value, Description);
        }
        #endregion
    }
}

```

Класс StateScheme

```

using System;

```



```

using System.Collections;

namespace AFC
{
    /// <summary>
    /// Схема состояния
    /// </summary>
    public class StateScheme : DescribedAutomatAttribute, ICloneable
    {
        #region Private members
        private IntArray m_Array;
        #endregion

        #region Public properties
        /// <summary>
        /// Набор идентификаторов выходных воздействий,
        /// выполняемых при переходе в это состояние
        /// </summary>
        public IntArray Actions
        {
            get
            {
                return m_Array;
            }
        }
        #endregion

        #region Constructor
        /// <summary>
        /// Создает схему состояния
        /// </summary>
        /// <param name="y">Идентификатор состояния</param>
        /// <param name="description">Описание состояния</param>
        /// <param name="inf">Набор идентификаторов выходных воздействий,
        /// выполняемых при переходе в это состояние</param>
        public StateScheme(int y, string description, params int[] inf)
            : base(y, description)
        {
            m_Array = new IntArray(inf);
            m_Array.OnChanged += new
                ObjectChangedEventHandler(this.ArrayChanged);
        }
        #endregion

        #region Protected methods
        protected void ArrayChanged(ChangeEventRaiser sender)
        {
            Changed();
        }
        #endregion

        #region Public methods
        /// <summary>
        /// Создает эквивалентную себе схему
        /// </summary>
        /// <returns>Созданная схема</returns>
        public object Clone()
        {
            return new StateScheme(Value, Description, m_Array.ToArray());
        }
        #endregion
    }
}

```

Класс TransitionScheme

```
using System;
using System.Collections;

namespace AFC
{
    /// <summary>
    /// Схема перехода
    /// </summary>
    public class TransitionScheme : DescribedAutomatAttribute, ICloneable
    {
        #region Private members
        private IntArray m_Inf;
        private IntArray m_YFfrom;
        private int m_YTo;
        private ConditionNode m_ConditionTree;
        #endregion

        #region Public properties
        /// <summary>
        /// Набор идентификаторов событий, использованных в условии перехода
        /// </summary>
        public ICollection UsedEvents
        {
            get
            {
                return ConditionTree.UsedEvents;
            }
        }

        /// <summary>
        /// Набор идентификаторов входных переменных, использованных в условии
        ///перехода
        /// </summary>
        public ICollection UsedVariables
        {
            get
            {
                return m_ConditionTree.UsedVariables;
            }
        }

        /// <summary>
        /// Флаг, определяющий, является ли переход групповым
        /// </summary>
        public bool IsGroupTransition
        {
            get
            {
                return m_YFfrom.Count>1;
            }
        }

        /// <summary>
        /// Набор идентификаторов выходных воздействий, выполняемых при
        ///переходе
        /// </summary>
        public IntArray Actions
        {
            get
            {

```

```

        return m_Inf;
    }
}

/// <summary>
/// Набор идентификаторов начальных состояний
/// </summary>
public IntArray YFrom
{
    get
    {
        return m_YFfrom;
    }
}

/// <summary>
/// Идентификатор конечного состояния
/// </summary>
public int YTo
{
    get
    {
        return m_YTo;
    }
    set
    {
        m_YTo = value;
        Changed();
    }
}

/// <summary>
/// Строковое представление условия
/// </summary>
public string Condition
{
    get
    {
        return Description;
    }
    set
    {
        Description = value;
        m_ConditionTree = new ConditionNode(Description);
        Changed();
    }
}

/// <summary>
/// Представление условия в виде бинарного дерева
/// </summary>
public ConditionNode ConditionTree
{
    get
    {
        return m_ConditionTree;
    }
}

#endregion

#region Constructor
/// <summary>
/// Создает схему группового перехода

```

```

/// </summary>
/// <param name="t">Идентификатор перехода</param>
/// <param name="yfrom">Набор идентификаторов начальных
/// состояний</param>
/// <param name="condition">Строковое представление условия</param>
/// <param name="yto">Конечное состояние</param>
/// <param name="inf">Набор идентификаторов выходных воздействий,
/// выполняемых при переходе</param>
public TransitionScheme(int t, int[] yfrom, string condition, int yto,
params int[] inf)
    :base(t, condition)
{
    m_YFfrom = new IntArray(yfrom);
    CheckYFrom();
    m_YFfrom.OnChanged += new
    ObjectChangedEventHandler(this.MemebersChanged);
    m_YTo = yto;
    Condition = condition;
    m_Inf = new IntArray(inf);
    m_Inf.OnChanged += new
    ObjectChangedEventHandler(this.MemebersChanged);
}

/// <summary>
/// Создает схему перехода
/// </summary>
/// <param name="t">Идентификатор перехода</param>
/// <param name="yfrom">Идентификатор начального состояния</param>
/// <param name="condition">Строковое представление условия</param>
/// <param name="yto">Конечное состояние</param>
/// <param name="inf">Набор идентификаторов выходных воздействий,
/// выполняемых при переходе</param>
public TransitionScheme(int t, int yfrom, string condition, int yto,
params int[] inf)
    : this(t, new int[]{yfrom}, condition, yto, inf)
{
}
}
#endregion

#region Protected methods
protected void MemebersChanged(ChangeEventRaiser sender)
{
    Changed();
    CheckYFrom();
}
#endregion

#region Private methods
private void CheckYFrom()
{
    if (m_YFfrom.Count == 0)
        throw new InvalidYFromException(m_YFfrom.ToArray(),
        "Некорректное значение начальных массива состояний");
}
#endregion

#region Public properties
/// <summary>
/// Создает эквивалентную себе схему
/// </summary>
/// <returns>Созданная схема</returns>
public object Clone()
{

```

```

        return new TransitionScheme(Value, m_Yffrom.ToArray(),
            Condition, m_YTo, m_Inf.ToArray());
    }
    #endregion
}
}

```

Класс VariableScheme

```

using System;

namespace AFC
{
    /// <summary>
    /// Схема входной переменной
    /// </summary>
    public class VariableScheme : DescribedAutomatAttribute, ICloneable
    {
        #region Constructor
        /// <summary>
        /// Создает схему входной переменной
        /// </summary>
        /// <param name="x">Идентификатор входной переменной</param>
        /// <param name="description">Описание входной переменной</param>
        public VariableScheme(int x, string description)
            : base(x, description)
        {
        }
        #endregion

        #region Public methods
        /// <summary>
        /// Создает эквивалентную себе схему
        /// </summary>
        /// <returns>Созданная схема</returns>
        public object Clone()
        {
            return new VariableScheme(Value, Description);
        }
        #endregion
    }
}

```

Приложение 2. Метод, реализующий переход автомата

```

/// <summary>
/// Метод, реализующий переход автомата
/// </summary>
/// <param name="e">Идентификатор события</param>
/// <returns>true - если переход был выполнен, в противном случае false</returns>
protected virtual bool run(int e)
{
    if (m_Scheme != null)
    {
        ICollection transitions = m_Scheme.FindTransitionsFromY(Y);
        foreach(TransitionScheme transition in transitions)
    }
}

```

```

    {
        foreach (ILogger logger in m_Loggers)
            logger.TransitionCheckingStarted(this, transition.Value);
        if (CalculateCondition(transition.ConditionTree, e))
        {
            foreach (ILogger logger in m_Loggers)
                logger.TransitionRealizationStarted(this,
                    transition.Value);
            foreach (int z in transition.Actions)
                Z(z);
            foreach (ILogger logger in m_Loggers)
                logger.TransitionRealizationFinished(this,
                    transition.Value);
            y = transition.YTo;
            StateScheme newstate = m_Scheme.FindState(transition.YTo);
            if (newstate.Actions.Count > 0)
            {
                foreach (ILogger logger in m_Loggers)
                    logger.StateEnteringStarted(this, y);
                foreach (int z in newstate.Actions)
                    Z(z);
            }
            return true;
        }
        else
        {
            foreach (ILogger logger in m_Loggers)
                logger.TransitionCheckingFailed(this,
                    transition.Value);
        }
    }
    return false;
}
return false;
}
}

```

Приложение 3. Интерфейсы «вычислителя входных пересенных» и «выполнителя выходных воздействий»

Интерфейс IInputVariableCalculator

```

using System;

namespace AFC
{
    /// <summary>
    /// Вычислитель входных переменных
    /// </summary>
    public interface IInputVariableCalculator
    {
        /// <summary>

```

```

    /// Указывает, может ли данный вычислитель работать с указанной
    /// переменной
    /// </summary>
    /// <param name="x">Идентификатор входной переменной</param>
    /// <param name="automat">Автомат</param>
    /// <returns>true - если возможно вычисление, false - в противном
    /// случае</returns>
    bool ImplementsVariable(int x, Automat automat);

    /// <summary>
    /// Вычисляет входную переменную
    /// </summary>
    /// <param name="x">Идентификатор входной переменной</param>
    /// <param name="automat">Автомат</param>
    /// <param name="eventContext">Контекст события</param>
    /// <param name="automatContext">Контекст автомата</param>
    /// <returns>Значение вычисленной переменной</returns>
    bool CalculateVariable(int x, Automat automat, object eventContext,
        object automatContext);
}
}

```

Интерфейс IOutputActionPerformer

```

using System;

namespace AFC
{
    /// <summary>
    /// Выполнитель выходного воздействия
    /// </summary>
    public interface IOutputActionPerformer
    {
        /// <summary>
        /// Указывает, может ли данный исполнитель работать с указанным
        /// воздействием
        /// </summary>
        /// <param name="z">Идентификатор воздействия</param>
        /// <param name="automat">Автомат</param>
        /// <returns>true - если выполнение возможно, false - в противном
        /// случае</returns>
        bool ImplementsAction(int z, Automat automat);

        /// <summary>
        /// Вычисляет выходное воздействие
        /// </summary>
        /// <param name="z">Идентификатор выходного воздействия</param>
        /// <param name="automat">Автомат</param>
        /// <param name="eventContext">Контекст события</param>
        /// <param name="automatContext">Контекст автомата</param>
        void PerformAction(int z, Automat automat, object eventContext, object
            automatContext);
    }
}

```

Приложение 4. Методы класса *Automat* для работы с

«ВЫЧИСЛИТЕЛЯМИ» И «ВЫПОЛНИТЕЛЯМИ»

```
/// <summary>
/// Добавляет вычислителя входной переменной
/// </summary>
/// <param name="input"></param>
public void AddInput(IInputVariableCalculator input)
{
    if (input != null)
        m_Input.Add(input);
    foreach (ILogger logger in m_Loggers)
        logger.InputAdded(this, input);
}

/// <summary>
/// Удаляет вычислителя входной переменной
/// </summary>
/// <param name="input"></param>
public void RemoveInput(IInputVariableCalculator input)
{
    m_Input.Remove(input);
    foreach (ILogger logger in m_Loggers)
        logger.InputRemoved(this, input);
}

/// <summary>
/// Добавляет выполнителя выходного воздействия
/// </summary>
/// <param name="output"></param>
public void AddOutput(IOutputActionPerformer output)
{
    if (output != null)
        m_Output.Add(output);
    foreach (ILogger logger in m_Loggers)
        logger.OutputAdded(this, output);
}

/// <summary>
/// Удаляет выполнителя выходного воздействия
/// </summary>
/// <param name="output"></param>
public void RemoveOutput(IOutputActionPerformer output)
{
    m_Output.Remove(output);
    foreach (ILogger logger in m_Loggers)
        logger.OutputRemoved(this, output);
}
```